Jaccard Coefficients as a Graph Benchmark

***Abstract*—There are presently several graph benchmarks in the literature, some with hundreds of published processing reports. They all, however, have several characteristics that make them of academic, but not necessarily real-world interest. This paper follows up on a proposed "Jaccard" graph benchmark with real-world relevance that involves the "neighborhoods" of vertices. Variations in algorithms and implementations are discussed, with an emphasis on a reference implementation and possible scaling via parallelism.**

***Keywords*-Graphs algorithms, Jaccard coefficients, Scalability**

I. Introduction

A graph is of a set of objects (vertices), and links (edges) between pairs of objects that represent some sort of relationships. Computing over such graphs is of increasing importance to a wide spectrum of application areas ranging from "conventional" communication and power networks, transport, and scheduling, to emerging applications such as social networks, medical informatics, genomics, and cybersecurity.

While there are several current graph benchmarks, some with hundreds of reported implementations, most of them are based on "academic" graph problems, and often have little direct value to real-world applications, especially when we want to understand the relative efficiency of different hardware architectures and configurations. Further, given that many of these graphs are growing in size, it is critical that we understand how to do such processing in parallel in an efficient manner. A companion paper [1] suggests three alternatives selected to overcome the issues with the current suite. These include computation of the overlap in neighbors between pairs of vertices, determination of a set of edges that form matchings between vertices, and stateful random walks.

This paper focuses on the first: computation of **Jaccard Coefficient** on very large graphs. The Jaccard coefficient $\Gamma(u, v)$ between two *entities* u and v (both vertices in a graph) is the ratio of the number of neighboring vertices common to both u and v to the total number of neighbors of either u and v. In the real world, such a number gives a measure of the "strength" of some "similarity" or "relationship" between the two entities. Common examples studied in the literature involve authors and documents, authors and reviewers, and actors and movies. Another application discussed in [2] is the similarity of Wikipedia users that edit Wikipedia pages, or symmetrically the similarity of web pages that are edited by the same user. Of more widespread relevance is in recommendation systems where a high overlap in what customers buy what products may indicate what other products to advertise on a per customer basis.

Another complex example from a real insurance problem [3] may be determining with whom a person has shared a residence. Consider a graph with several classes of vertices including at least people and residence addresses. An edge between a person and an address implies that that person once resided at that address. A relationship between two people u and v may be declared if the Jaccard Coefficient based on common residences is high. A "1.0" implies the two have always shared residences; a "0" implies they have never shared a residence; intermediate values implies different degrees of sharing. Extra weighting may be given for people who held the same last name during the residency.

In addition to the kernel, a graph benchmark with real-world relevance should represent accurately how the graph is represented and provided to the underlying implementation. A *batch* implementation must read in something akin to the edge list of a graph and generate all possible coefficients in the graph. This corresponds to the way most graph benchmarks are defined today. An *in-memory* implementation instead may start with a graph already constructed as a directly accessible data structure, and may be more relevant to real-world applications that want to perform a variety of on-demand analytics. Two versions of this are relevant here; one where a query requests all coefficients within the graph as in a batch implementation, and one where the coefficients for only a specific subset of vertices (perhaps only one) is requested. In either case, implementations must be capable of handling multiple concurrent queries. A third implementation version of particular real-world relevance is one that *streams* updates into an in-memory graph, and requests a list of what coefficients may change as a result.

This paper defines a formal benchmark for the core of such problems based on Jaccard coefficients. The rest of this paper is organized as follows. Section II introduces the problem formally and defines metrics. Section III discusses possible algorithms when all coefficients are desired. Section IV discusses streaming implementations. Section V discusses variations and heuristics that may help performance. Section VI overviews related work. Section VII discusses a sequential reference implementation. Section VIII discusses expression of the problem in a variety of graph programming paradigms. Section IX discusses parallel implementations and their expected scaling characteristics. Section X concludes. The Appendix lists some alternative implementations. Two companion papers [4], [5] cover the other two benchmarks proposed in [1] in a format similar to this paper.

II. Problem and Metrics

A *Jaccard Coefficient* (also called a *Jaccard Index* or a *Tanimoto Index*) is a measure of the "similarity" between two vertices (termed " entities") of a graph in terms of the overlap of their neighboring vertices (which need not be of the same class).[1] If $N(u)$ is the set of vertices that form the "neighborhood" of the vertex u (i.e. the set of vertices that have an edge from u to them), then the Jaccard coefficient Γ(u, v) between two vertices u and v is the ratio of the number of neighboring vertices in common to the total number of distinct neighboring vertices[2]:

$$\Gamma(u,v) = |N(u) \cap N(v)| \ / \ |N(u) \cup N(v)| \qquad (1)$$

An alternative representation of the above equation is based on two measures: γ(u, v) as the number of vertices in common between vertices u and v, and $d_{out}(u)$ as the *out-degree* of vertex u (the number of edges leaving u to other vertices).

$$\Gamma(u,v) = \gamma(u,v) \ / \ (d_{out}(u) + d_{out}(v) - \gamma(u,v)) \qquad (2)$$

Also, to better reflect many real world problems, we assume without loss of generality the graphs are bi-partite, that is there is a "left" set of vertices L and a "right" set of vertices R, with all edges E from a vertex in L to a vertex in R. The γs are then computed over pairs of vertices from L whose neighbors are from R. We call each such pair (u,v) where (u,w) and (v,w) are both in E, a "**link**". It is possible to have up to $O(|L|(|L|-1)/2$ non-redundant links in a graph[3]

The size of the sets L and R may be radically different. The problem from [3] has L as a set of people, and R a set of possible residential addresses. A "normal" problem in the 2012 time-frame had $|L| = 800$ million and $|R| = 100$ million

If $|L|$ is the number of vertices in L, then there can be up to $|L|(|L|-1)$ non-zero coefficients[4]. While $O(|L|^2)$ is potentially extremely large for real problems, real-world graphs seem to have an exponent of more like $|L|^k$ for k in the range 1.2 to 1.4. In any case, the number of non-zero coefficients will have a definite effect on the complexity of Jaccard algorithms.

Also affecting complexity is how the coefficients are to be saved and reported out. This may range from requiring storing all $|L|^2$ of them, to saving just the non-zeros, to just the non-zeros but where they may be streamed out in some arbitrary order. The last case implies that once a coefficient has been computed, it is not necessary to keep it around.

If we know $\gamma[u,v]$ then if we also know the out degrees $d_{out}(v)$ if each vertex, then the computation of the corresponding $\Gamma(u,v)$ is two adds and a divide. It is also the case that for many real-world problems, knowing $\gamma(u,v)$ is sufficient to provide the insight needed. That is certainly the case for [3]. Finally, we note that $\gamma(u,v) = \gamma(v,u)$, and thus if vertices can be placed in some numerical order (such as their index in an array) then we need only compute $\gamma(u,v)$ where $u < v$. This saves approximately 1/2 the computation. Thus this paper focuses on just computing the non-redundant γs.

In terms of performance metrics, the computation of links in some sense corresponds to the basic work involved in the computation of non-zero, non-redundant, γs as the final product. When divided by the execution time, this former gives us work in **links per second**, and the latter number gives us an overall performance metric for algorithms of "**JAC/s**", or "**JAccard Coefficients per Second**".

III. All-Coefficients Algorithms

Algorithms 1 through 4 represent algorithms for computing all γs in a graph with varying time and space complexity. Table I summaries upper bounds for these complexities. Note that the space complexity refers only to the memory needed to do the computations, not to save any γs at the end.

Algorithm 1 Jaccard via Adjacency Matrices:

L	=	set of "left" vertices
R	=	set of "right" vertices
E	=	$\{(u,v)|$ u in L, v in $R\}$
A	=	$|L|$ by $|R|$ matrix where $A[i,j] = 1$ if (i,j) in E

```
1: procedure J1(A)
2:     for $u$ in $L$ do
3:         for $v > u$ in $L$ do
4:             $\gamma[u,v] \leftarrow inner\_product(A[u,:], A[v,:])$
5:         end for
6:     end for
```

Algorithm 1 is the simplest. It assumes the graph is expressed as an adjacency matrix A where $A[u,v] = 1$ indicates there is an edge from vertex u to vertex v. Its time complexity is potentially cubic - an inner product is performed for each of the non-redundant (u,v) pairs, even the ones that end up being zero. For very dense graphs, this inner product involves up to $O(|R|)$ "multiply-adds." For the more realistic very sparse graphs, if sparse kernels are used, the work of each inner product is more proportional to the out-degrees of the vertices. In our prior notation, each "multiply" of two non-zeros is the formation of a link. Finally, note that this whole operation is equivalent to $\gamma \leftarrow AA^T$ where A^T is the transpose of A.

As expensive as this time complexity is, the working space complexity is essentially constant. Other than loop counts and indices, nothing needs to be carried over from one $\gamma(u,v)$ to the next.

Algorithm 2 is similar in that there is an $O(|V|^2)$ outer loop looking at all possible links, but instead of an inner product, for each w reachable from u, a search is made on the neighbors of v. A match corresponds to a link, and causes the corresponding $\gamma(u,v)$ count to be incremented. Time complexity is slightly higher because we cannot rely on the intrinsic "order" of edges the way we can with adjacency matrices. Working memory though is still constant.

If a higher memory complexity is permissible, lower time complexity is possible. Fig. 3 keeps a worst case $O(|L|)$

[1]The original *Jaccard Similarity* was a measure of the statistical similarity between two sets. The definition here makes each such set an "entity" vertex, with edges to the members of the set.

[2]$\cap$ stands for set intersection, $\cup$ is set union, and |...| denotes set cardinality

[3]The links (u,v) and (v,u) are the same; we count (u,v) where $u < v$ as the non-redundant one.

[4]Consider the case where each vertex in L has an edge to the same vertex in R.

Algorithm 2 Incremental Jaccard:
$L,\ R,\ E$ as above
$N(u)\ =\ \{w|(G) \text{ in } E\}$

```
1: procedure J2(U, V)
2:     for $u$ in $L$ do
3:         for $v\ >\ u$ in $L$ do
4:             $\gamma[u,v]\ \leftarrow\ 0$
5:             for $w$ in $N(u)$ do
6:                 if $w$ in $N(v)$ then
7:                     $\gamma[u,v] +=\ 1$
8:             end for
9:         end for
10:    end for
```

Algorithm 3 Matching Jaccard:
$L,\ R,\ E$ as above
γ is a vector of counts where $\gamma[v]$ is the count of all links from the current u.

```
1: procedure J3(G)
2:     for $u$ in $L$ do
3:         Initialize $\gamma$ to a vector of $|L|$ 0s
4:         for $w$ in $N(u)$ do
5:             if $(v,w)$ in $E$ then
6:                 if $u < v$ then $\gamma[v] += 1$
7:         end for ▷ All $\gamma$s involving $u$ have been computed
8:     end for
```

structure γ indexable by an L vertex, and for each u in L, it explores one at a time each neighbor of u, $N(u)$, in R. For each of these ws, a backwards exploration is performed of vertices v in L that have w as a neighbor. If $u < v$ then this is a non-redundant link and the associated γ is incremented. This is best done if each vertex in R has an N^{-1} function which indicates which vertices in L have edges to this one. Perhaps a smarter data structure than a simple indexable $O(|R|)$ vector is an (key,value) hash table where the key is a v and the value is the (u,v) count. Note also in this case, no γs are available until the end of the inner-most loop, when the whole row of γs for the same u are done. If these values need not be saved, the γ vector could then be reused for the next u.

Algorithm 4 achieves a potentially significantly reduced time complexity by performing a two-step join-like operation. Here we keep two hash table-like structures, and iterate first not over L vertices but over edges. For each edge (u,w) the u vertex is added to a list $\rho[w]$ associated with w. The second step then iterates through the non-empty buckets of w, and for each generates all possible (u,v) link pairs. Each pair then indexes into γ and increments the appropriate term. While the worst case memory complexity is $O(|L|^2)$, using hash tables can keep this latter structure down to something proportional to just the actual number of non-zero coefficients. This is the algorithm used in several real-world Jaccard-like computations (cf. [3]), with multiple level hash tables used to reduce the

Algorithm 4 Join Jaccard:
$L,\ R,\ E,\ \gamma$ as above
ρ is a pool of key-value pairs where keys are vertices from R and the value is a set of vertices from L

```
1: procedure J4(E)
2:     Initialize $\rho$, $\gamma$
3:     for $e\ =\ (u,w)$ in $E$ do
4:         Append u to $\rho[w]$
5:     end for
6:     for $w$ in $\rho$ do
7:         for all pairs $u,v$ in $\rho[w]$ do
8:             if $u < v$ then $\gamma[(u,v)] +=\ 1$
9:         end for
10:    end for
```

Algorithm	Time Complexity	Space Complexity
J1	$O(\|L\|^2 d_{out-max})$	$O(1)$
J2	$O(\|L\|^2 d^2_{out-max})$	$O(1)$
J3	$O(\|L\| d_{out-max} d_{in-max})$	$O(\|L\|)$
J4	$O(E) + O(\|R\| d^2_{in-max})$	$O(\|L\|^2)$
$d_{out-max}$ is maximum out degree of any vertex in L d_{in-max} is maximum in degree of any vertex in R		

TABLE I: Complexities.

search time in the hash tables to essentially near constant, even for large graphs.

While all of these algorithms compute all coefficients associated with a graph, Algorithms 1 through 3 explore parts of a graph and thus are most appropriate for in-memory implementations. Algorithm 4, however, builds up its data structures by scanning edges, with no particular order. As such, it is a particularly good candidate for batch implementations.

Further, Algorithms 1 and 2 have an outer loop over vertices u in L. Their inner loops represent decent algorithms looking for the coefficients associated with a particular u. Algorithm 2's inner loops go further, and represent algorithms to query the coefficient for a particular (u,v).

IV. Streaming

A final real-world variation may involve building algorithms that are **streaming**, that is some aspect of the graph changes with time. This may include the incremental addition or removal of vertices or edges. Given that adding a vertex adds no new γs, and a vertex cannot be removed until all edges to it have been removed, we will concentrate only on edge addition and removal.

A. Edge Addition

When we add an edge (x,w), there are $N^{-1}(w)$ *new* links generated. The resulting links may be using x in either the u or v position. Each of these links may also either update a previously non-zero γ or start a new one. If all prior $gamma$s are still accessible, then only $N^{-1}(w)$ updates to the γs need be made. If, however, there are too many non-zero γs to be kept around, then we need to recompute then from scratch. Algorithm 6 diagrams this case. Again it is probably a good implementation to construct the computed γs as an expandable set of (key,value) pairs.

Not included in Algorithm 6 is code to see if $x < y$ or vice versa.

NEEDS MORE DISCUSSION

Algorithm 5 Incremental Edge Addition:
$N(x)$, x in L, is the set of neighbors of x from R
$N^{-1}(w)$ is the set of vertices in L that have w as a neighbor in R.

```
1: procedure J5(EDGE(X, W))
2:     Initialize $\gamma$ to size $|N^{-1}(w)|$
3:     for $y$ in $N^{-1}(w)$ do
4:         $\gamma[x, y] += 1$
5:         for $z$ in $N(y)$ do
6:             if $z$ in $N(x)$ then $\gamma[x, y] += 1$
7:         end for
8:     end for
```

B. Edge Removal

NEEDS WRITEUP

V. Variations and Heuristics

A. Bloom Filters

A suggestion in [6] is to use a variation of a Bloom filter[5] to quickly identify pairs of L vertices that have no possibility of overlapping neighborhoods in R. To do this we partition R into M non-overlapping subsets of vertices $\{R_1, ...R_M\}$, with M typically a power of 2. For each x in L we construct an M-bit bit vector $b[x]$ where bit i is set to 1 if any edge (x, w) has its w in R_i.

Now given two vertices u and v from L, if the logical AND of $b[u]$ and $b[v]$ is all zeros, then there are no common neighbors. Using this in algorithm 2 would not reduce the worst case complexity, but could significantly reduce the time spent in the inner loop testing neighborhoods. It would not help algorithms 3 or 4 as they only explore pairs known to have common neighbors.

B. Thresholds

An observation from real world applications is that very often there are thresholds for either Γ or γ below which a Jaccard coefficient need not be reported. This is certainly the case for the example problem in [3].

The Bloom filter of the prior section can be modified to quickly provide an upper bound on a $\gamma(u, v)$ without a detailed exploration. Define the function $ones(b)$ as returning the number of 1s' in the bit vector b. The value returned from the expression $ones(b[u]\ AND\ b[v])$ is the minimum number of neighbors in common, and is the basis for the test in the prior section. Instead consider $ones(b[u]\ AND\ NOT(b[v]))$. This is the minimum number of neighbors of u that are known to not be in the neighborhood of v. If we subtract this from $d_{out}(u)$ (the out-degree of u) we get the maximum number of neighbors of u that could be neighbors of v. This is thus an upper bound on $\gamma(u, v)$. The symmetric expression $d_{out}(v) - ones(b[v]\ AND\ NOT(b[u]))$ gives another upper bound from v's perspective.

The smaller of these thus is a hard upper bound on $\gamma(u, v)$, and if it is less than the desired threshold for γ, (u, v) can be skipped. Likewise, a constant amount of computation can yield a hard upper bound on $\Gamma(u, v)$, and threshold as appropriate. This can be done without exploring neighborhoods, and thus could accelerate both algorithms 3 and 4.

C. Neighbors-Only Coefficients

The algorithms of the last section find Jaccard coefficients between all vertex pairs between sets L and R in a bipartite graph. Thus there are potentially as many as $|L| * |R|$ non-zeros. A variation which may be useful when $L = R$ is to look only at pairs that have an edge between them to begin with. In this case the maximum number of non-zero coefficients is $|E$ which is liable to be far smaller than $|L| * |R|$.

[7] describes an algorithm for this case, and gives some limited scaling results for up to 12 threads. In addition, this algorithm includes a threshold as discussed in Section V-B.

VI. Related Work

VII. Reference Implementation

NEEDS WRITEUP

A. Test Cases

A generator for bipartite graphs has been built in Python that allows separate specification of the size of L and R as well as the average degree. It then uses R-MAT, an algorithm based on that used in the Graph500 benchmark [8], but with edges only from L to R, and a separate set of probabilities for both the source vertices and the destination vertices. The original R-MAT algorithm produces oscillations within the degree distribution, and is not guaranteed to be monotonically decreasing. Random noise was added to the probabilities during the graph generation which has been shown to dampen these oscillations [9], producing distributions similar to those seen in real-world graphs. This allows for a somewhat more realistic distribution of edges.

NEEDS MORE DISCUSSION

VIII. Alternative Representations

A. MapReduce

The MapReduce framework, as popularized in the Hadoop environment [10], processes in parallel large data sets represented as "(key,value)" pairs. Processing occurs in a series of "steps," each of which has two phases. In a *Map* phase, each input pair is processed in some way into another (key,value) pair. In a *Reduce* phase, all pairs with the same key are aggregated in some way to produce yet another (key,value).

An early MapReduce computation of Γs is described in [2]. The problem solved has L as a set of set names, and R is the set of possible elements. The initial set of edges are equivalent

[5] c.f. https://en.wikipedia.org/wiki/Bloom_filter

to (key,value) pairs where the "key" is a set name X from L, and the "value" the name Y from R of an element of the set X. Processing proceeds in three MapReduce steps:

- Reproduce the input, but where each value has a second component equalling the out-degree of the set name.
- For each Y element, generate a pair where the key is empty and the value is a set of pairs: the name of a set that included it and the set name's out-degree.
- For each of the above pairs, the Map generates a set of pairs where the key is all possible pairs of set names, and the value the sum of the out-degrees. The Reduce then groups up all pairs where the key pair is the same and the value is the set of counts, and then uses this to compute the associated Γ.

A larger MapReduce study can be found in [11]. Here the graph used is not bipartite but a general single vertex class, undirected edge, RMAT graph as found in the Graph500 benchmark. The algorithm has 5 steps, with the middle 3 a cross between algoriths 2 and 3. An initial step annotated vs with degree, and a final step "randomized" the output. Problems of different sizes up to 64 million vertices (1 trillion edges) were run on a 1000 dual-socket node cluster. For the largest sized problems the system computed 149 million JACs per second.

IX. Scaling and Parallelism

Parallelism comes in three forms: shared memory multi-threading in an environment where all threads can see all the data structures, distributed memory multi-processing where the data structures are partitioned across different *nodes* (and explicit "messages" need be sent for interactions between them), and hybrid codes where both are employed. OpenMP is a typical programming environment for the former, and MPI for the middle.

Shared memory multi-threading Algorithms 1 and 2 is straightforward; teams of different worker threads can take on different u iterations concurrently. Algorithms 3 and 4 are somewhat more complex in that there is more irregular creation of child threads, and more synchronization needed for updating γs when links are discovered.

Distributed memory parallel implementations are more complex, especially as problems get more sparse and/or irregular. The Hadoop implementations discussed in Section VIII-A are examples. Further, as shown in [12], the computation of very sparse problems where the reduction in computation time due to the parallelism is quickly overwhelmed by communication costs, to the point where overall performance actually degrades with increasing parallelism.

A. A Partition-based Algorithm

An alternative parallel algorithm that avoids some of these issues can be derived from an algorithm described in [13] for counting triangles. Assume that L and R are partitioned into N and M subsets respectively. Then assume we partition the edges E into *edge blocks* $E_{i,j}$ that contain all, and only those edges from a vertex in L_i to R_j, as pictured in Fig. 1.

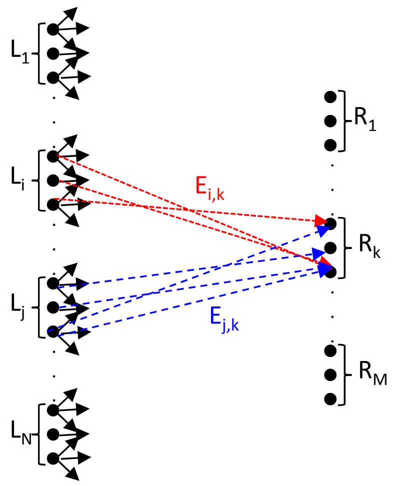

Fig. 1: Partitioned Bipartite Graph.

N/M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
3	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96
4	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160
5	15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240
6	21	42	63	84	105	126	147	168	189	210	231	252	273	294	315	336
7	28	56	84	112	140	168	196	224	252	280	308	336	364	392	420	448
8	36	72	108	144	180	216	252	288	324	360	396	432	468	504	540	576
9	45	90	135	180	225	270	315	360	405	450	495	540	585	630	675	720
10	55	110	165	220	275	330	385	440	495	550	605	660	715	770	825	880
11	66	132	198	264	330	396	462	528	594	660	726	792	858	924	990	1056
12	78	156	234	312	390	468	546	624	702	780	858	936	1014	1092	1170	1248
13	91	182	273	364	455	546	637	728	819	910	1001	1092	1183	1274	1365	1456
14	105	210	315	420	525	630	735	840	945	1050	1155	1260	1365	1470	1575	1680
15	120	240	360	480	600	720	840	960	1080	1200	1320	1440	1560	1680	1800	1920
16	136	272	408	544	680	816	952	1088	1224	1360	1496	1632	1768	1904	2040	2176

Fig. 2: Number of Edge Blocks for N,M Partitions.

Now consider the pair of edge blocks $E_{i,k}$ and $E_{j,k}$. This pair contains all the information needed to compute all links between u and v where u is in L_i, v is in L_j, and the links are through vertices in R_k. Any of the prior algorithms can be adapted to find these links, but have a greatly reduced in or out-degree to sort through. Further, there are NM different $E_{i,k}$s, and for each $E_{i,k}$ there are $N-(i-1)$ possible $E_{j,k}$s that don't have redundant links. Thus there are $MN^2/2 + MN/2$ separate pairs of edge sets, all of which may be performed in parallel, with only the updates due to each link needing to be done in an atomic, coordinated way across nodes. Algorithm 5 outlines such an approach, ignoring the needed distribution of edge sets. Fig. 2 lists the number of edge blocks for several N, M combinations. With this approach there are NM pairs of blocks involving L_1, but only M pairs involving L_N.

One of the issues with conventional parallel graph benchmark implementations is "hot spot" or "load imbalance" due to "heavy" vertices with far out of the ordinary in or out degrees. If edges are evenly distributed, then each block $E_{i,k}$ has about E/NM edges. Further, if edges are not evenly distributed, then the max out-degree for any L vertex in an edge block is at most $|R|/M$, and/or the maximum in degree of any R vertex in an edge block is at most $|L|/N$. In the worst case an edge block may contain $|L_i| * |R_k|$ edges, which if all partitions are of equal size is of order $|L| * |R|/NM$. If this

Algorithm 6 Partitioned Jaccard:

```
procedure J5(u, v)
    for all $1 \leq i \leq N$ do
        Initialize $\gamma$ to —L—/M x —R—
        for all $1 \leq k \leq M$ do
            for all $i \leq j \leq N$ do
                Find all links $(u, v)$ where $u$ in $L_i$,
                $v$ in $L_j$, and link goes thru $R_k$.
                For each link: $\gamma[u, v] += 1$
            end for
        end for
    end for
```

isn't sufficient to prevent load imbalance, then there is nothing in this approach that demands equal sized partitions. Thus a very heavy L vertex could be a L partition in its own right, or a very heavy R vertex could likewise be a separate R partition.

If there are $O(MN^2)$ processors each handling a different pair of edge sets, then the time of the innermost loop on one processor is approximately the time for the whole problem. If we use a variant of Algorithm 4, then a reasonable estimate of execution time would be $O((|R|/M)(d_{in-max}/N)^2)) \approx O((|R|d^2_{in-max})/MN^2$. This is $O(MN^2)$ times faster than Algorithm 4, which with $O(MN^2)$ processors is very nearly perfect strong scaling[6].

Note also that if we use a variant of Algorithm 4, we need not generate the cross-product of all the $E_{i,k}$ and $E_{j,k}$s intermingled, but only those from $E_{i,k}$ against those from $E_{j,k}$, which is a much smaller number. Further, when we know that $j > i$, then no redundant links are needed, and we can exclude a test of $v > u$. However, the case $j = i$, which is needed for links within an L partition, still must be checked.

Algorithm 5 also does not include the initialization of any needed γs or ρs. If we execute all pairs in parallel, then it is possible to generate links between any u and any v at any time. This requires a worst-case γ to be initialized. If, however, the outer loop is not done in parallel, but sequentially, then an initialization between steps 3 and 4 of a γ of only size $|L|/N$ by $|R|$ is needed.

Also if an Algorithm 4 variant is used for the inner code, then a ρ of only length $|R|/M$ need be built, but a separate such ρ is needed for each parallel instance.

Algorithm 7 outlines such a code. The code inside the outermost loop can be executed for each $E_{i,k}, E_{j,k}$ pair concurrently. The $if\ u < v$ need only be executed when $i = j$ as for all other pairs we are guaranteed that $u < v$

One of the key advantages of this algorithm, especially for distributed memory systems, is that given that a pair $E_{i,k}, E_{j,k}$ is positioned on a single node, then all the memory accesses to them are local. The only non-local references are the atomic updates to $\gamma[u, v]$.

[6]The $O(|E|)$ term in the time complexity only goes down by MN, so that may limit scalability for large parallelism.

Algorithm 7 Partitioned Parallel Join Jaccard:

```
procedure J7()
    Initialize $\gamma$
    for all combinations of $E_{i,k},\ E_{j,k}$ do
        Initialize $\rho_L,\ \rho_R$
        for all $(u, w)$ in $E_{i,k}$ do
            Append u to $\rho_L[w]$
        end for
        for all $(v, w)$ in $E_{j,k}$ do
            Append v to $\rho_R[w]$
        end for
        for all $w$ in $R_k$ do
            for all $u$ in $\rho_L[w]$ do
                for all $v$ in $\rho_R[w]$ do
                    if $u < v$ then
                        $\gamma[u.v] += 1$
                end for
            end for
        end for
    end for
```

X. Conclusions

Acknowledgment

This work was supported in part by NSF grant CCF-1642280, and in part by the University of Notre Dame.

Appendix

A. GraphBLAS

References

[1] P. M. Kogge, N. V. Chawla, D. Thain, B. A. Page, and N. A. Butcher, "Realistic computationally stressing graph benchmarks," In Preparation, 2018.

[2] J. Bank and B. Cole, "Calculating the jaccard similarity coefficient with map reduce for entity pairs in wikipedia," Wikipedia Similarity Team, Dec. 2008.

[3] P. Kogge and D. Bayliss, "Comparative performance analysis of a big data nora problem on a variety of architectures," in *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, 2013, pp. 22–34.

[4] P. M. Kogge, N. V. Chawla, D. Thain, B. A. Page, and N. A. Butcher, "Stateful random walks as a graph benchmark," In Preparation, 2018.

[5] ——, "Jaccard coefficients as a graph benchmark," In Preparation, 2018.

[6] P. M. Kogge, "Jaccard coefficients as a potential graph benchmark," *2016 IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, vol. 00, pp. 921–928, 2016.

[7] J. Scripps and C. Trefftz, "Parallelizing an algorithm to find communities using the jaccard metric," in *Proceedings of the 2015 IEEE International Conference on Electro/Information Technology*. IEEE, May 2015, pp. 4:1–4:8.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *In In Fourth SIAM International Conference on Data Mining*, 2004.

[9] C. Seshadhri, A. Pinar, and T. G. Kolda, "An in-depth analysis of stochastic kronecker graphs," *J. ACM*, vol. 60, no. 2, pp. 13:1–13:32, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2450142.2450149

[10] T. White, *Hadoop: the Definitive Guide*. O'Reilly Media, 2011.

[11] P. Burkhardt. (2014, Feb.) Asking hard graph questions: Beyond watson: Predictive analytics and big data. Beyond Watson Workshop. [Online]. Available: http://www.pdl.cmu.edu/SDI/2013/slides/big_graph_nsa_rd_2013_56002v1.pdf

[12] B. Page and P. M. Kogge, “Scalability of hybrid sparse matrix dense vector (spmv) multiplication,” in *accepted for High Performance Computing Symp. (HPCS)*, ser. HPCS’18, July. 2018.

[13] S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW ’11. New York, NY, USA: ACM, 2011, pp. 607–614. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963491