SINA: A Scalable Iterative Network Aligner

Abdurrahman Yaşar, Bora Uçar, Ümit V. Çatalyürek

SINA: A Scalable Iterative Network Aligner

Abdurrahman Yaşar
Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: ayasar@gatech.edu

Bora Uçar
Univ Lyon, CNRS, ENS de Lyon,
Inria, UCBL 1, LIP UMR 5668,
F-69007 Lyon, FRANCE
Email: bora.ucar@ens-lyon.fr

Ümit V. Çatalyürek
Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: umit@gatech.edu

***Abstract*—Given two graphs, network alignment asks for a potentially partial mapping between the vertices of the two graphs. This arises in many applications where data from different sources need to be integrated. Recent graph aligners use the global structure of input graphs and additional information given for the edges and vertices. We present SINA, an efficient, shared memory parallel implementation of such an aligner. Our experimental evaluations on a 32-core shared memory machine showed that SINA scales well for aligning large real-world graphs: SINA can achieve up to $28.5\times$ speedup, and can reduce the total execution time of a graph alignment problem with 2M vertices and 100M edges from 4.5 hours to under 10 minutes. To the best of our knowledge, SINA is the first parallel aligner that uses global structure and vertex and edge attributes to handle large graphs.**

I. INTRODUCTION

Fusion of data from different sources can be modeled as merging graphs. In such a case, given two non-isomorphic and comparable graphs, the first step is to identify a *graph alignment*, where a potentially partial mapping of vertices between two graphs is computed. Graph alignment is a well studied area. However, an important portion of related works are proposed for structurally very similar networks, such as protein protein interaction graphs [11], [26], [15], [9]. Furthermore, the majority of the existing work rely on computationally heavy optimization algorithms or pairwise similarity computations for all vertex pairs which cannot produce results for large graphs in a reasonable time. Hence, they are limited to graphs having less than tens of thousands vertices.

Most of the existing graph alignment techniques are based on the global topology of underlying graphs. More recent ones exploit additional information about vertex and edges for improved alignment, when such information is available. One recent approach is implemented in GSANA [27] which uses global global structure of the graphs to reduce the problem size. GSANA has been shown to have higher recall than the state of the art algorithms while being orders of magnitudes faster. However, for larger graphs (a few million vertices and edges), its execution time is still in the orders of hours. In this work, we investigate an efficient and scalable parallelization of GSANA without sacrificing its recall. Our aim is to enable high recall alignment of very large graphs on shared memory systems. We implement the findings in a data-parallel, architecture (resource)-aware scalable graph alignment framework called SINA.

GSANA is an iterative algorithm, where each iteration has four pipelined steps. In the first step, shortest path distances to a set of special vertices (called anchor vertices) are computed. Anchor vertices are those for which a mapping is known. If such vertices are available in the given graph, then GSANA uses them; if not, it decides which vertices to use as anchor vertices. Then some of these anchor vertices are singled out and deemed more important. In the second step, GSANA places vertices into a 2D plane (unit circle to be more precise) using the shortest path distances to the important anchor vertices, with the idea that similar vertices will be placed closely. The 2D plane is subdivided into buckets using quadtrees in the third step. In the fourth step, the similarities of vertices that lie in the same or neighboring buckets are computed by combining a set of similarity measures, and a promising set of pairs is mapped greedily. The next iteration then starts with an enlarged set of anchors, where the additional anchors are chosen with the computed mapping. In SINA we parallelize all costly components of GSANA.

The contributions of this paper are as follows:

- We develop parallel algorithms to implement SINA, making the process scalable, and solve the attributed network alignment problem, so that large scale graphs can be handled in a reasonable time.
- We propose vertex layout techniques for large scale graphs. These techniques are aimed at decreasing the total execution time by increasing the data access locality.
- We optimize GSANA's sequential implementation. SINA's sequential execution times are about 33 times faster than those reported in [27].

Experimental results show that our proposed framework, SINA, reaches up to $28\times$ speedup. To the best of our knowledge, SINA is the first parallel aligner that uses global structure and vertex and edge attributes. Therefore we could not compare SINA with other systems. We note that as SINA retains the recall of GSANA, it has higher recall than other state of the art methods.

The organization of the paper is as follows. We introduce the notation and give the formal problem definition in Section II. Then we discuss the parallelization approach in Section III. Section IV presents experimental results, which is followed by a summary of related work in Section V. We conclude the paper in Section VI with a summary and a list of future work.

Symbol	Description
V	Vertex set
E	Edge set
T_V	Vertex type set
T_E	Edge type set
$t[x]$	Type of the vertex $x \in V$ or the edge $x \in E$
A_V	Vertex attribute set
A_E	Edge attribute set
$a[x]$	Attribute of the vertex $x \in V$ or the edge $x \in E$
$N_i[u]$	Neighbor list of vertex u in graph G_i
$\delta(u, v)$	Distance between $u, v \in V$
$\sigma(u, v)$	Similarity score for $u \in V_1$ and $v \in V_2$
$\mu[u]$	Mapping of $u \in V_1$ in V_2
$S = S_1 \cup S_2$	Anchor (seed) set where $S_2 = \{v : \mu[u] = v, v \in S_1\}$

TABLE I
NOTATIONS USED IN THIS PAPER.

II. PROBLEM DEFINITION AND BACKGROUND

A graph $G = (V, E, T_V, T_E, A_V, A_E)$ consists of a set V of vertices, a set E of edges, two sets T_V and T_E for vertex and edge types, and two sets A_V and A_E for vertex and edge attributes, where type and attribute sets can be empty. For instance in a Facebook graph, *human, page, group* can be used as vertex types, *name, location* can be defined as vertex attributes, and connection types and connection years can be defined as edge types and attributes respectively. An edge e is referred as $e = (u, v) \in E$, where $u, v \in V$. The neighbor list of a vertex $u \in V$ is defined as $N[u] = \{v \in V : (u, v) \in E\}$. When discussing two graphs, we will use subscripts 1 and 2 to differentiate them if needed, and ignore those subscript when the intent is clear from the context. For example, $N_1[u]$ and $N_2[u']$ will represent the neighbor lists of vertices u and u' in G_1 and G_2, respectively. Given a vertex $x \in V$ or an edge $x \in E$, $a[x]$ represents the set of attributes of x, and $t[x]$ represents the type of x. We also use $\delta(u, v)$ to denote the breadth-first search (BFS) distance between vertices u and v. We assume that there are a number of vertices whose mappings are known. These are called anchor vertices, where the initial set of anchor vertices is denoted by S, which can be empty.

Given two different graphs G_1 and G_2, the similarity score between two vertices $u \in V_1$ and $v \in V_2$ is denoted by $\sigma : V_1 \times V_2 \to \mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers. We define $\mu[u] : V_1 \to V_2$ as an injective mapping, where $\mu[u] = v$ represents mapping of $u \in V_1$ to $v \in V_2$. If a vertex $u \in V_1$ is not mapped to a vertex of G_2, we use $\mu[u] = \perp$, which is also refereed as **nil** mapping. Table I displays the notations used in this paper.

Definition 1 (Graph Alignment Problem)**.** Given two graphs $G_1 = (V_1, E_1, \dots)$ and $G_2 = (V_2, E_2, \dots)$, the graph alignment problem asks for an injective mapping that maximizes

$$\sum_{\forall u \in V_1, \mu[v]} \sigma(u, \mu[u]) \ , \tag{1}$$

with the convention that $\sigma(u, \perp) = 0$, when $\mu[u] = \perp$.

GSANA, hence SINA, uses *quadtrees* [10] to partition the 2D plane, hence the vertices of the graphs. Simply put, quadtree is a geometric, tree-based data structure in which each internal node has four children. The points in the plane (in our case the vertices) are stored in the leaf nodes, where a capacity limit is imposed in the number of points in a leaf node. Initially there is a single node, containing the whole plane. Then, the plane is recursively split into four regions with a cross until each of the regions contains at most a predetermined number of points. When a region is split, the node representing that region becomes an internal node, and each of the four sub-regions are associated with a leaf node whose parent is the new internal node. The points that lie in the region are partitioned among the four leaf nodes according to their positions.

III. SINA: PARALLEL ITERATIVE NETWORK ALIGNMENT

The sequential GSANA algorithm is iterative and has four main steps: i) shortest path computations, ii) anchor selection, iii) partitioning, and iv) mapping (see Fig. 1). We parallelize all compute intensive parts which are marked in Fig. 1. We discuss the sequential versions of each of these steps as implemented in GSANA below, and then discuss SINA's parallelization approach in the following subsections.

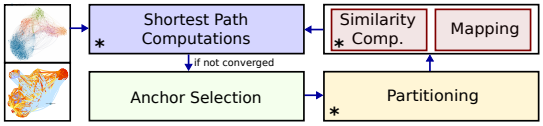


Fig. 1. Overview of the SINA, where boxes marked with "*" are parallelized.

GSANA uses shortest path distances to anchor vertices to reduce the similarity computation requirements. In simple terms, if we know that $v_1 \in V_1$ is identified with $u_1 \in V_2$, then $v_2 \in V_1$ is more likely to be mapped to $u_2 \in V_2$ than to $u_3 \in V_2$, if $\delta(u_2, u_1)$ is closer to $\delta(v_2, v_1)$ than $\delta(u_3, u_1)$. GSANA assigns unit lengths to edges so that the shortest path distances effectively correspond to the breadth-first search distances. We discuss the parallelization approach of SINA for this first step in Subsection III-A.

When there are a number of different anchor pairs, one has to combine the distances of a vertex pair to the anchor vertices to decide if the two vertices could be similar. GSANA identifies some anchors as better fit for determining similar vertices (these better serving anchor vertices are called vantage anchors in GSANA). Then, GSANA places the vantage anchor vertices on the unit circle in 2D, where an anchor pair is placed diametrically opposite places on the circle. Once the vantage anchors are placed in the circle, the other vertices are placed inside this circle using the distances to the anchors on the circle. Since nearby vertices will be checked for similarity, GSANA divides up this circle into buckets and confines the similarity computations among neighboring buckets. In order to have similar number of vertices per bucket, GSANA inserts the vertices in a quadtree (with an upper bound on the number of vertices per a leaf node of the quadtree) while placing

them in the 2D plane. Instead of using one quadtree, SINA uses one quadtree per graph. We needed this deviation from GSANA for better definition of tasks for parallelism (covered in Subsection III-C). We discuss the parallelization approach of SINA for the anchor selection step in Subsection III-B.

Once the vertices are inserted into the buckets corresponding to the leafs of the quadtree, GSANA starts computing similarity between vertices that lie in a common or a neighboring bucket. Vertices in buckets that are not neighbors (even if they are geometrically close by) are not compared, as the alternatives in the neighboring buckets are closer. We discuss the parallelization approach of SINA for the similarity computation in Subsection III-C.

When similarity computation is done, top k most similar vertices are identified for each vertex. Mapping is the fourth step of GSANA, and the goal is to compute potentially a partial mapping between two graphs using identified vertices. This step takes only a tiny fraction of the total execution time (less than 0.5s. per iteration), therefore we do not parallelize it. SINA implements a few optimization techniques, such as data layout and data structure optimizations for increasing practical performance. These are discussed in more detail in Subsection III-D.

A. Parallelizing shortest path computations

SINA needs an efficient breadth first search (BFS) implementation to compute the shortest path distances from each anchor vertex. There are two ways to do this. One of them is to use existing parallel [3] or vectorized BFS implementations [25] per BFS. The second one is to use a sequential BFS per anchor, and parallelize over anchors. Since the number of anchors is usually larger than the potential number of computing threads, yet relatively smaller with respect to the number of vertices (GSANA limits the number of anchors to 2000), this parallelization is more promising. In particular, the sequential tasks on a given graph (a BFS per anchor) have the same complexity and are likely to have similar practical run time. Furthermore, the parallelization overhead is negligible. Because of these reasons, SINA implements the second alternative described in Algorithm 1. As seen in this algorithm, for each newly introduced anchor $u \in G_1$, a sequential BFS is run starting from u in a parallel for. Once the shortest path distances from the newly introduced anchor vertices of G_1 are computed, a BFS for each newly introduced anchor vertex in G_2 is run in a parallel for. After BFSs, the new set of vantage anchors are determined as in GSANA. We did not parallelize this step of choosing vantage anchors, as its cost is negligible.

B. Placing vertices into 2D and bucketing

SINA needs to place all vertices in a graph into a quadtree. In the quadtree data structure, we keep vertices in the leaf nodes and use internal nodes for routing. Leaf nodes are split into four when the number of vertices in them reaches to a pre-defined size limit, l. In our partitioning, we use two main operations of quadtrees; search and insert. Since the search operation is read-only, it can be done concurrently. Therefore as the first step, in parallel, SINA computes all vertices' coordinates (p) in the $2D$ space and instantiates an insert operation, defined in Alg. 2. This algorithm first finds the corresponding leaf node for the given position p. When the leaf node is found, the insertion operation starts. The insertion operation requires some additional effort for two reasons. First, multiple threads may want to insert a vertex into the same queue at the same time. Second, when a leaf node has to be split, children should be created safely by one thread, and all the vertices in the leaf node have to be moved into the newly created children's queues. To overcome the first issue, we use thread-safe queues in leaf nodes, therefore multiple threads can make parallel updates. For the second issue, when a thread observes that a leaf node has to be split, it first locks the leaf node and creates four children. Then, the leaf node is marked as an internal node, and other threads cannot insert any more vertices to that node, or split that node. Finally, the vertices in the queue are moved to the children nodes. Here, we note that more than one thread may reach the split operation at the same time but only one of them creates the children. Then these threads may move the vertices in the queue in parallel.

Algorithm 1: Computation of the shortest paths from the anchor vertices with BFS

```
D_1.resize(|S|), D_2.resize(|S|)                    ▷ Memory allocation
▷ Computation of shortest paths from new anchors
for each u ∈ S_1 in parallel do
    if δ(u,.) is not computed before then
        δ(u,.) ← BFS(G_1, u)
for each u ∈ S_1 in parallel do
    if δ(μ[u],.) is not computed before then
        (δ(μ[u],.) ← BFS(G_2, μ[u])
```

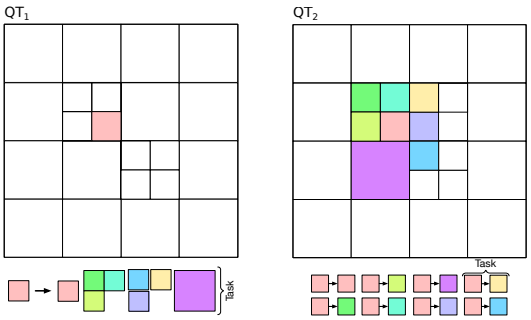
Algorithm 2: INSERT$(QRoot, u, p, l)$

```
Input : QRoot is a node in the quadtree, QT_i; u ∈ V_i
        and p is the 2D coordinates of u; l is the
        bucket size limit
if isLeaf(QRoot) then
    ▷ Push vertex u in position p, to the queue
    push(QRoot.B, u, p)          ▷ B is a concurrent queue
    if |QRoot.B| > l and isLeaf(QRoot) then
        lock(QRoot)
        if not hasChildren(QRoot) then
            createChildren(QRoot)
            markAsInternalNode(QRoot)
        unlock(QRoot)
        moveVertices(QRoot.B)
else
    ▷ For p, find the child
    quad ← findQuad(QRoot, p)
    insert(quad, u, p)
```

C. Similarity computation

In the fourth step, GSANA compares all vertices in a bucket of a graph with the vertices of the other graph that appear in the same or neighboring buckets. It checks neighbor buckets to make sure that vertices are close to the border of the buckets are handled appropriately. In SINA, we have two quadtrees. The first one contains the vertices of the first graph, and the second one contains those of the second graph. For each bucket $B_1 \in QT_1$, we define an associated one in QT_2 by locating the bucket $B_2 \in QT_2$ containing the center of B_1. Then, the similarity comparisons of vertices in $B_1 \in QT_1$ will be carried out for the vertices in the associated $B_2 \in QT_2$, and B_2's neighboring buckets in QT_2.

We propose two coarse-grain task definitions in SINA. In the *all comparison* scheme (ALL), the similarity computation between vertices in a bucket of QT_1 and all related buckets in QT_2 are defined as one large coarse-grain task. The second scheme, *pairwise comparison* (PAIR), is still a coarse task definition but it is finer than the first one. In this scheme, the similarity computation between the vertices in a bucket of QT_1 and one of the related ones in QT_2 is defined as a task. These two definitions are depicted in Figure 2, and explained in more detail below.



(a) ALL (b) PAIR

Fig. 2. Task definition and parallel computation schemes.

In ALL, a task is composed of similarity computations between the vertices of a non-empty bucket $B_1 \in QT_1$, and the vertices of the all the *related buckets* in QT_2. To find the related buckets, we first find the bucket $B_2 \in QT_2$ containing the center of B_1. The related buckets are B_2 and the neighbors of B_2.

In PAIR a task is composed of similarity computations between the vertices of a non-empty bucket $B_1 \in QT_1$, and the vertices of a single related bucket $B' \in QT_2$.

Alg. 3 constructs the task lists based on the selected scheme. Then, it executes each task in parallel using COMPSIMALL or COMPSIMPAIR algorithms displayed in Algorithms 4 and 5, respectively. These functions compute the similarity scores for each vertex $v \in B$ with every other vertex $u \in B'$. GSANA only keeps the top k most similar vertices that are identified for each vertex, and these are stored in a priority queue $P[v]$ for $v \in V_1$. The two similarity computation algorithms update those priority queues.

Algorithm 3: PARALLELSIM($QT_1, QT_2, scheme$)

Input : QT_1, QT_2 are quadtrees of G_1, G_2. $scheme$ is the comparison scheme
Output : $P[v]$: top k similar vertices of vertex $v \in V_1$ in V_2.

```
▷ Compute the task list T.
T ← ∅
for each non-empty B1 ∈ QT1 do
    B2 ← QT2.findLeafNode(B1.center)
    if scheme = ALL then
        B' ← B2 ⋃_{N∈QT2.neigbors(B2)} N
        T ← T ∪ {⟨B1, B'⟩}
    else
        for each B' ∈ QT2.neigbors(B2) do
            T ← T ∪ {⟨B1, B'⟩}
▷ Execute the task list T in parallel.
P[v] ← ∅, for ∀v ∈ V1
for each task T = ⟨B, B'⟩ in parallel do
    if scheme = ALL then
        COMPSIMALL(B, B', P)
    else
        COMPSIMPAIR(B, B', P)
return P
```

In COMPSIMALL (Alg. 4) all of the similarity scores for each vertex $v \in V_1$ will be computed by a single thread. Therefore, computed scores can be safely inserted to the priority queue $P[v]$ without any concurrency issues.

Algorithm 4: COMPSIMALL(B, B', P)

```
for each v ∈ B1 do
    for each u ∈ B' do
        compute σ(v, u)
        P[v].insert(u, σ(v, u))          ▷ Only keeps top k
```

In COMPSIMPAIR (Alg. 5), the top similarity scores for each vertex $v \in V_1$ might come from multiple threads. Therefore, for each task (i.e., for each bucket pairs that are compared), top k similar vertices first stored in a temporary queue, p. After top k similar ones computed for this task, if the highest similarity score in p is greater than the lowest similarity score in $P[v]$, then we lock the $P[v]$, insert all elements of p to $P[v]$ and finally unlock it. At any time, both p and $P[v]$ only keep just top k most similar vertices.

Algorithm 5: COMPSIMPAIR(B, B', P)

```
for each v ∈ B do
    p ← ∅                    ▷ Only keeps top k similar vertices for v
    for each u ∈ B' do
        compute σ(v, u)
        p.insert(u, σ(v, u))
    if p.top() > P[v].bottom() then
        lock(P[v])
        P[v].insertAll(p)            ▷ Only keeps top k
        unlock(P[v])
```

The ALL scheme has two main drawbacks. First, the number of parallel tasks is limited by the number of buckets. Second, due to irregular nature of partitioning and very coarse-

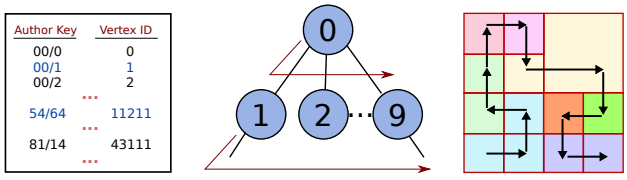
grain task composition, this scheme may lead to high load imbalance among tasks. In SINA (Alg. 3), tasks in the task list $\mathcal{T}$ are executed in parallel using a dynamic load balancing scheme (using OpenMP or CilkPlus). This will help toi reduce the observed load balance. Furthermore, one can simply sort tasks based on their loads in a non-increasing order, and use this largest-job first heuristic to further reduce the load imbalance, and hence the total execution time.

The PAIR scheme helps to reduce the load imbalance at the expense of additional synchronization overhead for insertion to the global priority queues. The use of local priority queues helps to reduce the synchronization overheads, but if two or more concurrent tasks have the same source bucket, this overhead may increase. To reduce the probability of such cases, SINA randomly shuffles the task list.

D. Other performance optimization techniques

We applied a few optimizations techniques to better exploit parallelism and improve data locality in SINA. Here, we cover these optimizations under two categories: data layout and data structure.

1) Data Layout: Exploiting data locality is one of the most common optimization techniques to improve the performance of irregular applications. The graphs that are input have already an ordering of the vertices, based on the alphabetical ordering of vertex labels. We call this the natural ordering (NAT). In SINA, we investigate two other layout techniques: "Breadth-First Search (BFS)" based and "Hilbert-Curve Based (HCB)" based. These techniques are presented in Fig. 3.



(a) NAT Layout (b) BFS Layout (c) HCB Layout

Fig. 3. Data layouts based on different vertex orderings.

a) BFS Layout: In this layout, shown in Fig. 3(b), the vertices are ordered based on their insertion time to the frontier queue in a BFS, starting from a vertex with the highest degree. This layout gives significant improvement in both similarity and shortest path computation steps over the natural layout. However in this scheme even if the vertices preserve some locality, appearances of the vertices in the quadtree buckets may not preserve BFS locality, which can cause loss of locality during similarity computation.

b) HCB Layout: In this layout, shown in Fig. 3(c), the vertices are ordered based on their Hilbert orders. When all vertices are inserted into the quadtree, SINA sorts the buckets based on a Hilbert order of the buckets. Then, SINA orders all vertices in a bucket according to the bucket's rank. We expect this layout to increase locality during the similarity computation step, where neighboring buckets are likely to be involved with the similarity computations regarding a given bucket in QT_1.

2) Data-Structure optimizations: The similarity function σ used in GSANA is composed of several metrics. All of these metrics (listed in Table II) measures similarity based on a different property.

TABLE II
COMPONENTS OF THE SIMILARITY FUNCTION σ.

Symbol		Description
τ	:	Type similarity
α	:	Anchor similarity
Δ	:	Relative degree distance [13]
τ_V	:	$\#Same/\#Total$ types of adjacent vertices
τ_E	:	$\#Same/\#Total$ types of adjacent edges
C_V	:	Vertex attribute similarity
C_E	:	Edge attribute similarity

All of the mentioned similarity metrics, except τ and Δ, require accessing two vertices' neighborhood information. For instance, to compute $\tau_V(u,v)$ a thread has to read $|N[u]|+|N[v]|$ different elements in T_V to be able to compare types of the neighboring vertices. This type of data access is highly irregular and cache unfriendly. Therefore to improve the cache use, SINA stores "weight" and "type" informations of the neighbors of a vertex in sorted vectors for each vertex.

IV. EXPERIMENTAL EVALUATION

We present several experiments in order to identify the performance trade-offs of the parallel algorithms and optimizations of SINA. Experiments were carried out on a machine that has two, 16-core Intel Xeon E5-2683 2.10GHz processors, 512GB of memory, 1TB disk space, running Ubuntu GNU/Linux with kernel 4.8.0. SINA is implemented in C++ and compiled with GCC 5.4.

A. Dataset

We use real-world graphs obtained from [4], [7], [22]. We also generated different size DBLP [22] graphs. The properties of graphs are listed in Table III, which are described below.

DBLP (2014-2017): As discussed in GSANA [27], we downloaded two consecutive years of DBLP graphs from 2014 [7] to 2017 [22]. The ground-truth is created using key attribute of author elements. Vertices are authors, and two authors have an edge if they have co-authored a paper. As in GSANA, we use publications' cross-ref information to create vertex attributes by splitting a cross-ref by '/' and unionizing initial character of each word as the vertex attribute. Edge attribute between two vertices is the mean of the publication years of co-authored papers between two authors. The other DBLP graphs (DBLP-17-small and DBLP-14-small), listed in Table III, are smaller subgraphs of the original DBLP graph, centered around the highest degree vertex.

Hollywood (2009-2011): Hollywood graphs from years 2009 and 2011 are downloaded from [4]. In these graphs, the vertices are actors, and two actors are connected if they have

acted in the same movie. We create the ground truth using author names. We give initials of each actor as the vertex attribute. This dataset does not have vertex/edge types nor edge attributes.

B. Workload

In order to investigate load imbalance due to variations of number of vertices in each bucket and variable degrees of the vertices, we plotted bucket and task workload distributions in Fig. 4. Here, the upper bound on the limit of vertices per bucket (i.e., leaf nodes of quadtree) is set to 512. Fig. 4(a) displays a histogram of number of buckets with different maximum number of vertices in a bucket. As expected, the vertex distribution among the buckets is not uniform.

A better estimate for workload for each task is the amount of data that needs to be read, at least once, in order to compute all required similarity scores. Computing similarity score between vertices v and u requires comparing the adjacency lists of those two vertices. Since we have multiple components of similarity scores, we need to multiply this number with a constant. For the sake of simplicity, we present the amount of data that needs to be read, in a single pass as

$$W(T = \langle B, B' \rangle) = \sum_{\forall v \in B} \sum_{\forall u \in B'} |N[v]| + |N[u]| \,. \quad (2)$$

Using the formula (2) as a metric for workload estimation, in Figure 4(b) and Figure 4(c) we present workload estimations for ALL and PAIR parallelization schemes respectively. In both of these figures x-axis represents tasks in execution order, and y-axis represents size of memory read in MB. We observe from these figures that in PAIR, tasks have better workload since the granularity sizes are smaller. In ALL, some tasks require significantly larger memory reads and hence higher computation times, yielding poor scalability.

C. Effect of Task Granularity and Parallel Programing API

In the next experiment, we compared ALL and PAIR parallelization schemes using two different shared memory programming languages/API: OpenMP and CilkPlus. In order to highlight load imbalance issues, in this experiment we have used our smallest problem (DBLP-17-small and DBLP-14-small graphs). Note that the sizes of these graphs are larger than the graphs used in prior studies. Figure 4(d) displays the speedup obtained using these four combinations. When we use static scheduling in ALL, OpenMP achieves only around $14\times$ speedup (not shown), and CilkPlus can only reach to approximately $12\times$ speedup. Using dynamic scheduling in OpenMP increases the speedup to only around $20\times$. In PAIR scheme and using dynamic scheduling SINA obtains around $26\times$ speedup with OpenMP and around $25\times$ speedup with CilkPlus. This demonstrates that, both of the programming models have competitive performances in PAIR. We observe that even though PAIR incurs additional synchronization overhead, it outperforms ALL, thanks to its finer granularity which yields better load balancing. We again note that in an attempt to reduce the number of locks in PAIR, SINA shuffles the task list before the execution. Since PAIR outperforms ALL, in the remaining experiments we use PAIR as the default scheme.

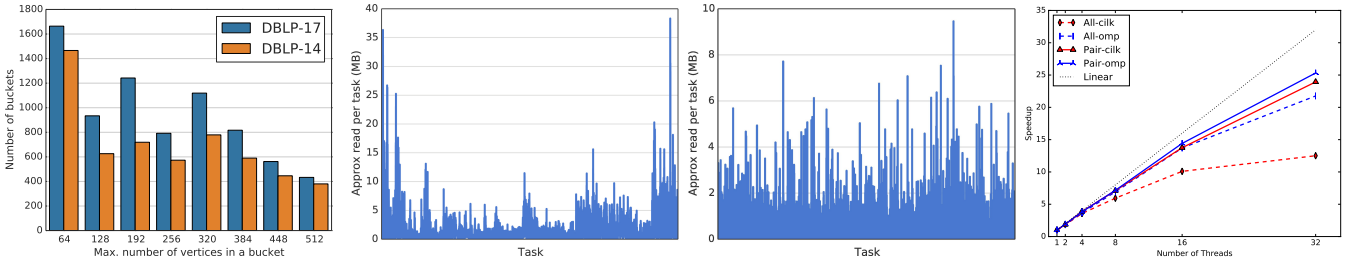
D. Data Layouts and Speedup

Next we compare three different data layouts obtained using three different vertex ordering on DBLP 14-17 dataset in Fig. 5. Figure 5(a) illustrates the execution time of SINA for this problem for three different data layouts and varying number of threads. In this figure, each grouped stacked-bar in the x-axis represents the number of threads and y-axis represents the execution time of the three parallel components of SINA. Sequential portion of SINA takes less than 10 seconds, hence it is omitted in this figure for simplicity. In the sequential run, more than 75% of the overall execution time comes from similarity computation. When we use 32 threads, this ratio decreases to 60%, because the parallelizations of the other two components do not scale as much as the similarity computation. As expected NAT order has the worst performance, because in this layout SINA portrays a more irregular data access pattern. Both BFS and HCB layouts improve the overall execution time with respect to NAT. BFS and HCB layouts give improvements in different parts of SINA. We observe that in the shortest path computations, BFS layout outperforms HCB layout by nearly 20% while HCB outperforms BFS in the similarity computation by nearly 25%. Since most of the overhead comes from the similarity computation, in the overall case, for 32 threads HCB outperforms BFS by nearly 10% and outperforms NAT by 40%. Figure 5(b) presents overall relative speedup results for three different layouts. In this figure, blue color represents speedups based on each layout's sequential execution time. Speedup results represented with orange color are computed based on the minimum sequential execution time which is with HCB. When we compare each layout with its sequential execution time, NAT layout gives $25\times$, BFS gives $24\times$ and HCB layout gives $23\times$ speedup. NAT and BFS gives $14\times$ and $20\times$ speedup when we compare them with the minimum sequential time. Since HCB outperforms other two layout techniques, in the following experiments, we use HCB as the default data layout.

Figure 6 illustrates execution times and speedup on Hollywood 09-11 dataset. In Figure 6(a), we observe that similarity computation takes more than 85% of the total execution time. As listed in Table III even though the Hollywood graphs have nearly the same number of vertices as the DBLP graphs, the number of edges per vertex is much higher. This is the primary reason as to why similarity computation dominates the total execution time. Figure 6(b) illustrates the overall and the three components' speedup. We observe that in this dataset SINA achieves a speedup of $30\times$ and gets closer to the linear speedup for similarity computation, and achieves $28\times$ speedup for the overall execution. Even though the speedup of the shortest path and partitioning computations are far from the ideal at $14\times$ and $20\times$, respectively, the overall speedup is not affected significantly, since these two components take a small portion of the total execution time.

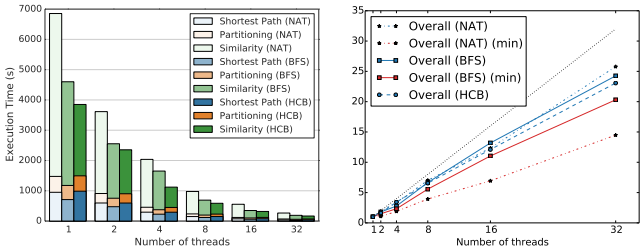
TABLE III
PROPERTIES OF THE DATASETS. $\langle|N[x]|\rangle$ REPRESENTS AVERAGE VERTEX DEGREE, AND $|\mu|$ REPRESENT THE SIZE OF GROUND TRUTH MAPPING.

Data Set	$\|V\|$	$\|E\|$	$\langle\|N[x]\|\rangle$	$\max(\|N[x]\|)$	$\|N[x]\| < 3$		$\|\mu\|$	$\|S_1\|$	A_V	A_E
DBLP-17-small	491,719	4,089,071	8.31	2,322	51,035	(10%)	294,531	64	✓	✓
DBLP-14-small	366,137	2,542,331	6.94	1,782	46,853	(13%)				
DBLP-17	1,966,877	9,059,634	4.61	2,322	616,386	(31%)				
DBLP-14	1,464,539	5,906,792	4.03	1,782	491,206	(34%)	1,440,379	64	✓	✓
Hollywood 2011	1,917,070	114,181,101	59.6	13,107	119,333	(6.2%)	990,090	64	✓	✓
Hollywood 2009	1,069,126	56,841,216	53.2	11,468	72,540	(6.8%)				



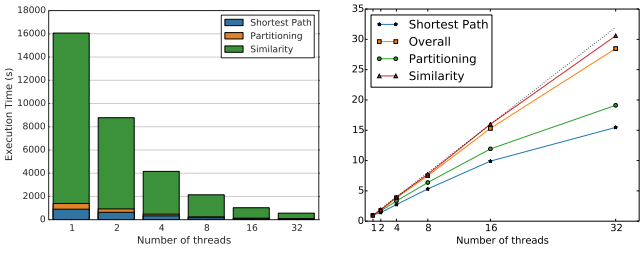
(a) Buckets (b) All (c) Pair (d) Speedup

Fig. 4. (a) Histogram of bucket sizes; (b) and (c) task workloads for ALL and PAIR, respectively; (d) overall speedup of different parallelization schemes using OpenMP and CilkPlus.



(a) Execution Time (b) Speedup

Fig. 5. Effect of data layouts to execution time and speedup in DBLP 14-17 dataset.



(a) Execution Time (b) Speedup

Fig. 6. Execution time and speedup for Hollywood 09-11 dataset using HCB data layout.

In Table IV we present speedups of the three parallel components and overall execution time for different datasets. For the similarity computation, SINA obtains 23× to 30× speedup, partitioning's speedup varies from 17× to 19×. One interesting fact that we observe is that for the shortest path computation while the speedup is between 22× to 23× in DBLP graphs, in Hollywood graphs this decreases to 15×. Again, since graphs are denser, frontier queues for BFS's enlarges faster and data access becomes more irregular.

TABLE IV
SPEEDUP AND OVERALL EXECUTIONS TIMES FROM 1 TO 32 THREADS.

Data Set		1	2	4	8	16	32
DBLP 14-17 (NAT)	Preparation	1.0	1.6	3.2	6.4	12.6	23.2
	Partitioning	1.0	1.7	3.2	6.1	11.9	18.8
	Similarity	1.0	2.0	3.4	7.3	12.4	27.4
	Overall	1.0	1.9	3.4	7.1	12.4	25.8
	Exec. time (s)	6854	3614	2038	978	555	266
DBLP 14-17 (BFS)	Preparation	1.0	1.5	3.1	6.1	11.6	22.0
	Partitioning	1.0	1.7	3.2	5.9	11.3	17.5
	Similarity	1.0	1.9	2.7	6.9	13.9	26.3
	Overall	1.0	1.9	2.7	6.7	13.2	24.3
	Exec. time (s)	4600	2553	1652	691	349	190
DBLP 14-17 (HCB)	Preparation	1.0	1.7	3.4	6.6	12.8	23.2
	Partitioning	1.0	1.7	3.2	6.3	11.6	19.1
	Similarity	1.0	1.6	3.5	6.6	12.0	24.2
	Overall	1.0	1.6	3.4	6.6	12.1	23.1
	Exec. time (s)	3851	2354	1121	588	318	167
Hollywood 09-11 (HCB)	Preparation	1.0	1.4	2.7	5.3	9.9	15.5
	Partitioning	1.0	1.7	3.3	6.4	11.9	19.1
	Similarity	1.0	1.9	4.0	7.7	16.1	30.6
	Overall	1.0	1.8	3.9	7.5	15.3	28.5
	Exec. time (s)	16064	8777	4158	2143	1050	564

V. RELATED WORK

Existing graph alignment methods are usually classified into four basic groups [6], [9]: spectral methods [16], [21], [23], [26]; graph structure similarity methods [1], [15], [18], [19], [20]; tree search or tabu search methods [5], [14], [17], [24]; and integer linear programming (ILP) methods [2], [8], [11]. All of these works have scalability issues. Our algorithms leverage global graph structure and reduce the problem space. Furthermore, with effective parallelization of each compute intensive step of the algorithm, we alleviate most of the scalability issues.

We are aware of two parallel approaches for global alignment. The first approach, termed network similarity decomposition (NSD) [12], decomposes the ranking calculations of IsoRank's similarity matrix [26] using the singular value decomposition. While NSD shows an order of magnitude improvement over the state of the art sequential algorithms, the largest network tested has less than ten thousand vertices. The second approach is a shared memory parallel algorithm [21] that is based on the belief propagation (BP) solution for integer program relaxation [2]. It uses parallel matrix operations for BP iterations and also implements an approximate weighted bipartite matching algorithm. Neither of these approaches can handle additional information (vertex and edge attributes and labels) available in modern graphs.

VI. Conclusion

We presented SINA, a scalable iterative graph aligner. SINA is a careful, shared memory parallelization of a recent sequential graph aligner GSANA [27]. SINA can achieve up to $28.5\times$ speedup on a 32-core machine, reduces the total execution time of a graph alignment problem with 2M vertices and 100M edges from 4.5 hours to under 10 minutes, while retaining the state of the art alignment recall obtained by GSANA.

As a future work, we will investigate how to use both BFS and HCB layouts at the same time. BFS layout is highly beneficial for the shortest path computations, while HCB layout is beneficial for the similarity computations. In order to improve the overall speedup, keeping two copies of the graphs in two different layouts (BFS layout for the shortest path computations, and HCB layout for the similarity computations) could be practical and useful. Another alternative is to use hierarchical ordering that combines HCB and BFS. We will also look at how to make use of existing shortest path distances when computing the shortest paths to the new anchor vertices, in order to reduce the run time both theoretically and practically.

References

[1] A. E. Aladağ and C. Erten, "Spinal: scalable protein interaction network alignment," *Bioinformatics*, vol. 29, no. 7, pp. 917–924, 2013.

[2] M. Bayati, M. Gerritsen, D. F. Gleich, A. Saberi, and Y. Wang, "Algorithms for large, sparse network alignment problems," in *IEEE International Conference on Data Mining (ICDM)*, 2009, pp. 705–710.

[3] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[4] P. Boldi and S. Vigna, "WebGraph Datasets: Laboratory for algorithmics," http://law.di.unimi.it/datasets.php, April 2018.

[5] L. Chindelevitch, C.-Y. Ma, C.-S. Liao, and B. Berger, "Optimizing a global alignment of protein interaction networks," *Bioinformatics*, vol. 29, no. 21, pp. 2765–2773, 2013.

[6] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[7] E. Demaine and M. Hajiaghayi, "BigDND: Big dynamic network data," http://projects.csail.mit.edu/dnd/, 2017.

[8] M. El-Kebir, J. Heringa, and G. W. Klau, "Lagrangian relaxation applied to sparse global network alignment," in *IAPR International Conference on Pattern Recognition in Bioinformatics*. Springer, 2011, pp. 225–236.

[9] A. Elmsallati, C. Clark, and J. Kalita, "Global alignment of protein-protein interaction networks: A survey," *IEEE Transactions on Computational Biology and Bioinformatics*, vol. 13, no. 4, pp. 689–705, 2016.

[10] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[11] G. W. Klau, "A new graph-based method for pairwise global network alignment," *BMC Bioinformatics*, vol. 10, no. 1, p. S59, 2009.

[12] G. Kollias, S. Mohammadi, and A. Grama, "Network similarity decomposition (nsd): A fast and scalable approach to network alignment," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 12, pp. 2232–2243, 2012.

[13] D. Koutra, H. Tong, and D. Lubensky, "Big-align: Fast bipartite graph alignment," in *IEEE International Conference on Data Mining (ICDM)*, 2013, pp. 389–398.

[14] S. Kpodjedo, P. Galinier, and G. Antoniol, "Using local similarity measures to efficiently address approximate graph matching," *Discrete Applied Mathematics*, vol. 164, pp. 161–177, 2014.

[15] O. Kuchaiev, T. Milenković, V. Memišević, W. Hayes, and N. Pržulj, "Topological network alignment uncovers biological function and phylogeny," *Journal of the Royal Society Interface*, vol. 7, no. 50, pp. 1341–1354, 2010.

[16] C.-S. Liao, K. Lu, M. Baym, R. Singh, and B. Berger, "Isorankn: spectral methods for global alignment of multiple protein networks," *Bioinformatics*, vol. 25, no. 12, pp. i253–i258, 2009.

[17] D. Liu, K. C. Tan, C. K. Goh, and W. K. Ho, "A multiobjective memetic algorithm based on particle swarm optimization," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 37, pp. 42–50, 2007.

[18] N. Malod-Dognin and N. Pržulj, "L-graal: Lagrangian graphlet-based network aligner," *Bioinformatics*, vol. 31, no. 13, pp. 2182–2189, 2015.

[19] V. Memišević and N. Pržulj, "C-graal: C ommon-neighbors-based global gra ph al ignment of biological networks," *Integrative Biology*, vol. 4, no. 7, pp. 734–743, 2012.

[20] T. Milenkovic, W. L. Ng, W. Hayes, and N. Przulj, "Optimal network alignment with graphlet degree vectors," *Cancer informatics*, vol. 9, p. 121, 2010.

[21] B. Neyshabur, A. Khadem, S. Hashemifar, and S. S. Arab, "Netal: a new graph-based method for global alignment of protein–protein interaction networks," *Bioinformatics*, vol. 29, no. 13, pp. 1654–1662, 2013.

[22] U. of Trier, "DBLP: Computer science bibliography," http://dblp.dagstuhl.de/xml/release/, 2017.

[23] R. Patro and C. Kingsford, "Global network alignment using multiscale spectral signatures," *Bioinformatics*, vol. 28, no. 23, pp. 3105–3114, 2012.

[24] V. Saraph and T. Milenković, "Magna: maximizing accuracy in global network alignment," *Bioinformatics*, vol. 30, no. 20, pp. 2931–2940, 2014.

[25] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Regularizing graph centrality computations," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 106–119, Feb 2015.

[26] R. Singh, J. Xu, and B. Berger, "Pairwise global alignment of protein interaction networks by matching neighborhood topology," in *Annual International Conference on Research in Computational Molecular Biology*, 2007, pp. 16–31.

[27] A. Yaşar and Ü. V. Çatalyürek, "An iterative global structure-assisted labeled network aligner," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2018.