Load Balanced Clustering Coefficients

Oded Green *
Georgia Institute of Technology
Atlanta, Georgia, USA

Lluís-Miquel Munguía
Georgia Institute of Technology
Atlanta, Georgia, USA

David A. Bader
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

Clustering coefficients is a building block in network sciences that offers insights on how tightly bound vertices are in a network. Effective and scalable parallelization of clustering coefficients requires load balancing amongst the cores. This property is not easy to achieve since many real world networks are scale free, which leads to some vertices requiring more attention than others. In this work we show two scalable approaches that load balance clustering coefficients. The first method achieves optimal load balancing with an $O(|E|)$ storage requirement. The second method has a lower storage requirement of $O(|V|)$ at the cost of some imbalance. While both methods have a similar time complexity, they represent a tradeoff between maintaining a balanced workload and memory complexity. Using a 40-core system we show that our load balancing techniques outperform the widely used and simple parallel approach by a factor of $3X - 7.5X$ for real graphs and $1.5X - 4X$ for random graphs. Further, we achieve $25X - 35X$ speedup over the sequential algorithm for most of the graphs.

Keywords

Parallel Algorithms, Graph Algorithms, Social Network Analysis, Scalable Programming

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph Algorithms*

1. INTRODUCTION

Clustering coefficients is a graph analytic that states how tightly bound vertices are in a graph [23]. The tightness is measured by computing the number of closed triangles in the graph, which can then imply the small-world property. Computing the clustering coefficients has been applied to many types of networks: communication [21], collaboration [22], social [22], and biological [4]. Clustering coefficients is used in a wide range of social network analysis applications. In such context, one can think of the local clustering coefficients as the ratio of actual mutual acquaintances versus all possible mutual acquaintances.

Clustering coefficients can be computed in two different variants: global and local. The global clustering coefficients is a single value computed for the entire graph, whereas the local clustering coefficients is computed per vertex. Both are computed in a similar fashion. Without the loss of generality, we consider the global clustering coefficients in this work, specifically when presenting pseudo-code. Nonetheless, our approach is applicable to computing local clustering coefficients as well. Table 1 presents the notations used in this paper.

We can formally define clustering coefficients as the sum of the ratios of the number of triangles over all possible triangles:

$$CC_{global} = \frac{1}{|V|} \sum_{v \in V} CC_v = \frac{1}{|V|} \sum_{v \in V} \frac{tri(v)}{deg(v) \cdot (deg(v) - 1)}$$

Clustering coefficients can be computed in multiple approaches [20]: enumerating over all node-triples, matrix multiplication, and intersecting adjacency lists. As many real world networks are considerably sparse, we focus on the last of these three approaches which has a time complexity of $O(|V| \cdot d_{max}^2)$ where d_{max} is the vertex with largest adjacency. The pseudo-code for this approach can be found in Algorithm 1. Many real world networks have a skewed vertex degree distribution which present parallel load balancing challenges.

In this work, we show that it is possible to estimate the total amount of work required by the clustering coefficients algorithm in $O(|E|)$ steps. While the load balancing may seem costly, for sparse graphs where $O(|E|) < O(|V| \cdot d_{max}^2)$. We show in Section 4 that this computation is negligible in time on an actual system for many real world sparse graphs.

We show two load balancing techniques: the edge-based approach and the vertex-based approach. These differ in the fact that the edge-based approach offers a better workload balance than the vertex-based approach. While this advantage is desirable, it comes

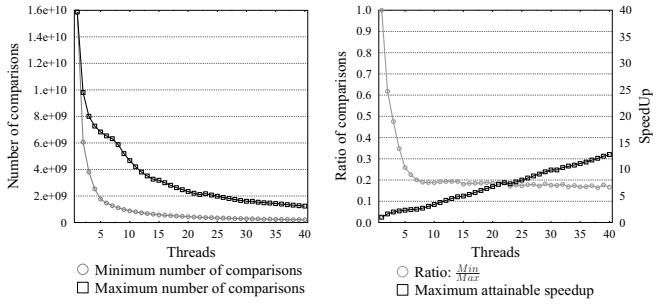
Algorithm 1: Serial algorithm for computing the number of triangles and clustering coefficients.

```
CC_global ← 0;
for v ∈ V do
    for u ∈ adj(v) do
        if u = v then
            next u;
        C ← intersect(v, adj(v), u, adj(u));
        CC_global ← CC_global + |C| / (|V|·deg(v)·(deg(v)−1));
```

*Corresponding author: ogreen@gatech.edu

PPAA'14, February 16, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2567634.2567635



(a) (b)

Figure 1: (a) Load distribution among parallel cores. (b) On the main ordinate, the ratio between the threads with minimum and the maximum number of comparisons is depicted. The secondary ordinate shows the maximum attainable speedup.

at an increased computational and memory space cost. The edge-based approach requires an $O(|E|)$ memory and $O(|E|)$ operations that evenly split the work to the p parallel cores. On the other hand, the vertex-based approach requires an $O(|V|)$ memory and $O(|E|)$ operations, which also are split among the cores. Both approaches can be executed in parallel.

The remainder of the paper will be structured as follows. This section discusses the challenges with computing clustering coefficients in parallel and briefly introduces our solutions. Section II discusses the related work and discusses real world graph properties and introduces vertex covers. In Section III, we present our two approaches for effective load balancing. In Section IV, we discuss our experimental methodology and present quantitative results. Finally, in Section V, we present our conclusions.

1.1 Parallel Clustering Coefficients Challenges and Solutions

In [3, 8, 12], it was shown that several real world networks follow a power law distribution on the number of adjacent edges to a vertex. As the time complexity of the algorithm is dependent on the square of the degree of the vertex with the highest degree, d_{max}, a simple division of the vertices amongst the cores such that each core receives an equal number of vertices is not likely to offer a good load balance. This is due to the fact that a single core might receive multiple high degree vertices. This can cause a single core to become the execution bottleneck. The focus of our work is to overcome this challenge.

Fig. 1 depicts an example of the load balancing issues caused by straightforward division schemes when computing clustering coefficients for a graph with a non-uniform adjacency distribution. Fig. 1 (a) plots the minimum and the maximum number of comparisons performed by the different parallel cores. Assuming that the total amount of work (number of comparisons on our case) required by the algorithm is known and is defined as $Work$, the maximal parallel speedup that can be attained for each algorithm is limited by the thread with maximum number of comparisons, $comp_{max}$:

$$max_{speedup} = Work/comp_{max}. \quad (1)$$

By this definition, uneven work distributions can affect scalability severely. The main ordinate of Fig. 1(b) depicts the ratio between the minimum and the maximum number of comparisons shown in Fig. 1(a). The secondary ordinate of Fig. 1(b) shows the maximal attainable speedup of the work distribution of the algorithm used to plot Fig. 1(a). In fact, it is the observation from above, that motivated the development of load balancing techniques that take into account the unique workload properties of clustering coefficients.

Table 1: Notations in this paper

Name	Description
CC_{global}	Global clustering coefficients
CC_v	Clustering coefficients for vertex v
$deg(v)$	Degree of vertex v.
$tri(v)$	Number of triangle that vertex v is in.
d_{max}	Vertex with maximal degree in the graph.
P and p	Number of parallel cores.
V	Set of vertices in a graph.
E	Set of edges in a graph.
u, v	Vertices in the graph.
t	Thread number.
$Vertex\_adj$	Array of size $\|V\|$ containing the starting positions of the vertex adjacencies
$eWork$	Array of size $\|E\|$ used to accumulate the work estimation of every connected vertex pair.
$vWork$	Array of size $\|V\|$ used to accumulate the work estimation of every vertex.
$Pivots$	Array of size $P+1$ that holds the starting and ending points of the work done by each core.
t	Thread id.
cc_t	Local thread clustering coefficients value.
cc_{global}	Global clustering coefficients value.

2. RELATED WORK

Clustering coefficients was first introduced by Watts and Strogatz [23]. Since, it has become a common means to quantify structural network properties. In essence, clustering coefficients measures the tightness of neighborhoods in graphs. They can be computed for both dense and sparse graphs, yet, they offer more insights for sparse graphs, many of which have the small-world property. The Small-World property was first presented by Milgram [18] and suggested that people in the United States can be related in less than six steps of separation.

An additional graph property of significance is the power-law distribution of edges in a network. Graphs featuring this characteristic have a large number of vertices with low degrees and a small number of vertices with high degrees, see by Faloutsos *et. al.* [12] and Barabási and Albert [3].

Clustering coefficients for a given graph is often reduced to enumerating the triangles formed between every triplet of vertices. Schank and Wagner [20] present an extensive review on other various serial algorithms along with performance comparisons over both "real world" networks and synthetic graphs. Still in the context of serial algorithms, Green and Bader [13] propose a novel clustering coefficients algorithm that employs vertex covers in order to reduce the number of list intersections and the number of actual comparisons needed to compute the triangle enumeration. We show that our load-balancing techniques can be extended to this algorithm as well.

The increase in the network size from thousands of vertices to millions and possible billions in the foreseeable future and the relevance of dynamic graphs has brought about a need for effective computations of clustering coefficients. The advances for faster clustering coefficients algorithms focus in parallelization techniques as well as approximation schemes. Both improvements are orthogonal concepts: while parallelization reduces the computation time, approximation can offer insights on the closeness of vertices when the cost of computing the clustering coefficients is prohibitive. Special techniques can also be developed for dynamic

graphs. Dynamic graph algorithms allow updating the analytic without doing a full recomputation every time the underlying network is modified. In practice, these optimizations are applied concurrently.

Several examples of algorithms using these optimizations can be found in the works of Becchetti *et al.* [6], Bar-Yossef *et al.* [2], and Buriol *et al.* [9] where triangle counting approximation techniques are employed on streaming graphs as a measure to cope with large data sets.

Ediger *et al.* [11] present both a parallel exact and parallel approximate algorithm for computing clustering coefficients for dynamic graphs. Their approach employs Bloom filters and they show results on the Cray XMT (a massively multithreaded architecture) for graphs with over a half a billion edges. Leist *et al.* [16] provide multiple parallel implementations using several GPUs and IBM Cell-BE processors for smaller graphs.

While these algorithms tackled many of the different aspects of computing clustering coefficients, they do not deal with the inherent load imbalance that is typical for many graph algorithms using static scheduling techniques. In such a scenario, the work is partitioned by the runtime and it is unaware of the properties of the application. As such, the application designer is responsible for dividing the work equally to the cores.

Both [5] and [17] consider the problem of workload imbalance for other graph algorithms. They use similar techniques to the ones that we use in this paper, which are based on prefix sum. Prefix sum is a basic primitive that can also be efficiently parallelized. A first straightforward parallel implementation was introduced by Hillis and Steele [15]. Blelloch [7] then showed a PRAM parallel work-efficient algorithm. In [14] a GPU implementation of the prefix sum is introduced.

In [5], a Breadth First Search algorithm is presented. The vertices in each level are partitioned to the multiple cores based on the sum of the adjacent vertices. A prefix sum is employed followed by a binary search in order to elaborate the partitioning. The overhead of these two partitioning stages is negligible compared to the remainder of the BFS. In [17], scalable GPU graph traversals are presented that partition the traversal edges equally among the multiple GPU streaming processors.

Other load balancing mechanisms can be used on specialized architectures. Ediger *et al.* [10] made use of the online scheduler of the massively multithreaded architecture of the Cray XMT for computing clustering coefficients. The scheduler is responsible for dispatching tasks when processors become available, thus achieving an effective parallelism. They show nearly perfect load balancing upto 64 XMT processors for several different graph types. Beyond the 64 processors, the speedup continues to grow but does not always scale perfectly - this is most likely due to workload imbalance.

3. LOAD BALANCED SCALABLE CLUSTERING COEFFICIENTS

We present two different techniques, which are conceptually similar and consist of two highly parallel phases. The first phase approximates the expected amount of work. With the work estimation at hand, we proceed to partition the work to the cores. This is followed by showing that each of these steps is itself balanced. In the second phase, the adjacency lists are intersected by the multiple cores using a modified version of Algorithm 1.

The workload estimation process is based off of Algorithm 1 and consists of foreseeing the amount of work needed to intersect vertices and edge endpoints. For that purpose, we employ two basic work estimations, which are discussed in depth. Both estimations can be defined in terms of the amount work needed for intersecting the adjacency lists of two vertices $u, v \in V$.

For simplicity, we assume that the adjacency lists are sorted. Given the sorted lists, the upper bound for the adjacency list intersection is $deg(u) + deg(v)$ comparisons. This number represents the worst-case scenario for a adjacency list intersection. An actual list intersection might be cut short if all the elements of one of the lists are traversed. However, this cannot be detected without doing the intersection or further testing. Nonetheless, we find $deg(u) + deg(v)$ to be a fair estimation of the adjacency list intersection.

We can define $Work(v, u)$ as the number of comparisons needed for the intersection of the adjacencies of v and u:

$$Work(v, u) = deg(v) + deg(u). \tag{2}$$

We proceed to gather this estimation for the vertex endpoints of every edge in the graph. This is formalized in terms of the previous definition as:

$$Work(G(V, E)) = \sum_{(u,v)\in E} Work(u, v) = \sum_{(u,v)\in E} (deg(v) + deg(u)) \tag{3}$$

Such definition can also be expressed in terms of vertices and their adjacencies:

$$Work = \sum_{(u,v)\in E} Work(u, v) = \sum_{v\in V} \sum_{u\in adj(v)} deg(v) + deg(u) \tag{4}$$

Specifically, the number of comparisons required for a specific vertex is:

$$Work(v) = \sum_{u\in adj(v)} (deg(v) + deg(u)) = deg(v)^2 + \sum_{u\in adj(v)} deg(u) \tag{5}$$

We derive the time complexity of clustering coefficients from (5). Computing either $\Sigma_{(v,u)\in E} Work(v, u)$ or $\Sigma_{v\in V} Work(v)$ allows to partition the work into near equal units to the multiple cores available. The first technique, which we refer to as the edge-based approach, requires $O(|E|)$ memory and theoretically offers optimal partitioning assuming that all comparisons are executed[1]. The vertex-based approach reduces the memory requirement to $O(|V|)$ at the expense of non-equal partitioning. In Section 4 we quantitatively compare these two approaches for real networks. While we have yet to discuss the algorithms in detail, we note that both algorithms have a similar upper bound time complexity, yet the accuracy of the partitioning will slightly change based on the storage complexity.

3.1 Edge-Based Approach

We present the first of our two approaches that partitions the work equally among the multiple cores. We refer to this method as the edge-based approach and its pseudo-code can be found in Algorithm 2. In Table 1, a reference is given for the variables used by our methods. Overall, we distinguish two distinct computation phases: 1) work estimation and load balancing and 2) clustering coefficients computation.

[1]As discussed earlier, a list intersection might complete early based on the actual adjacencies.

Algorithm 2: Edge-Based algorithm.

input : Graph $G(V, E)$, number of cores p
output: Clustering coefficients value cc_{global}
For $t \leftarrow 1$ *to* p **do in parallel**
  // Stage 1: Workload estimation
  $Pivots_t \leftarrow$ `BinarySearch` $(Vertex\_adj, t \cdot |E|/p)$;
  `SynchronizationBarrier()`;
  for $v \leftarrow Pivots_t$ *to* $Pivots_{t+1}$ **do**
    for $\forall u \in adj(v)$ **do**
      $eWork_{(v,u)} \leftarrow deg(v) + deg(u)$;
  `SynchronizationBarrier()`;
  `ParallelPrefixScan` $(eWork)$;
  $Pivots_t \leftarrow$`BinarySearch` $(eWork, t \cdot |E|/p)$;
  `SynchronizationBarrier()`;
  // Stage 2: CC calculation
  $cc_t \leftarrow 0$;
  $E_t \leftarrow$ all edges between $Pivots_t$ and $Pivots_{t+1}$
  for $e = (v, u) \in E_t$ **do**
    `VertexIntersection` $(v, deg(v), u, deg(u))$;
    $cc_t \leftarrow cc_t + \frac{|C|}{|V| \cdot deg(v) \cdot (deg(v)-1)}$;
  `SynchronizationBarrier()`;
  $cc_{global} \leftarrow$ `ParallelReduction`(cc_t);

Algorithm 3: Vertex-based algorithm.

input : Graph $G(V, E)$, number of cores p
output: Clustering coefficients value cc_{global}
For $t \leftarrow 1$ *to* p **do in parallel**
  // Stage 1: Workload estimation
  $Pivots_t \leftarrow$ `BinarySearch` $(Vertex\_adj, t \cdot |E|/p)$;
  `SynchronizationBarrier()`;
  for $v \leftarrow Pivots_t$ *to* $Pivots_{t+1}$ **do**
    $vWork_v \leftarrow 0$;
    for $\forall u \in adj(v)$ **do**
      $vWork_v \leftarrow vWork_v + deg(v) + deg(u)$;
  `SynchronizationBarrier()`;
  `ParallelPrefixScan` $(vWork)$;
  $Pivots_t \leftarrow$ `BinarySearch` $(vWork, t \cdot |E|/p)$;
  // Stage 2: CC calculation
  $cc_t \leftarrow 0$;
  for $v \leftarrow Pivots_t$ *to* $Pivots_{t+1}$ **do**
    $triangles \leftarrow 0$;
    for $\forall u \in adj(v)$ **do**
      `VertexIntersection` $(v, deg(v), u, deg(u))$;
      $triangles \leftarrow triangles + |C|$;
    $cc_t \leftarrow cc_t + \frac{triangles}{|V| \cdot deg(v) \cdot (deg(v)-1)}$;
  `SynchronizationBarrier()`;
  $cc_{global} \leftarrow$ `ParallelReduction`(cc_t);

Using an array of size $O(|E|)$ the expected number of comparisons for each edge is calculated using expression (2). The first step in Stage 1 divides the edges equally among the p cores such that each core receives $|E|/p$ edges. Assuming that the graph is given in a CSR representation, a binary search is conducted by each core into the vertex offset array. This vertex offset array is essentially a prefix sum array of the edge degrees in the graph. Hence, the binary search in the vertex offset array for the value $t \cdot |E|/p$ allows finding the vertex to which that edge belongs to, for a given thread $t \in \{1, 2, ..., p\}$.

Due to the fine grained load balancing, several cores may intersect adjacency lists for the same vertex[2]. For simplicity, we assume that the sets of adjacencies assigned to the different cores do not overlap. This is a fair assumption given adjacency distributions of many real world graphs. However, if there is a vertex with a large enough degree that it dominates the execution time, it is possible to modify the algorithm such that the adjacencies of a vertex is shared by multiple cores. Additionally, a single large adjacency intersection can be divided to several cores using a modified version of Merge Path [19] which is parallel merging algorithm.

Each core will compute $Work(v, u)$ for the set of edges it has been assigned. The results will be stored in the $Work$ array of size $O(|E|)$. Once this is completed, a parallel prefix sum is computed on this array in order to obtain the total amount of work computed from the first vertex up to the current vertex. The last entry in the prefix array maintains the expected number of comparisons for the entire graph. Upon completion of the prefix sum, an additional binary search is executed per core into the prefix sum array. The binary search finds the partitioning pivots of the algorithm. The binary search might actually want to divide a specific list intersection to multiple core. As discussed before, for simplicity our algorithm does not partition the work to multiple cores. In reality, this is not a concern and is discussed in Section 4. When the binary search is completed, the partitioning pivots for clustering coefficients are available and we can proceed to compute the clustering coefficients as part of Stage 2.

3.2 Vertex-based approach

We refer to our second load balancing technique as the vertex-based method. Its pseudo-code can be found in Algorithm 3. This approach reduces the storage requirement from $O(|E|)$ to $O(|V|)$ by performing the load balance at a vertex granularity. As a result, some imbalance might be introduced and a single vertex can become a bottleneck of the algorithm. We will see in Section 4, that this imbalance does not reduce the total performance of the vertex-based approach. In comparison with the edge-based approach, this imbalance is minute.

Similarly to the previous technique, the load-balanced clustering coefficients calculation is comprised of two main stages: 1) the workload estimation and 2) the clustering coefficients calculation.

The amount of work is calculated using expression (5). In the first stage, each thread performs a binary search of the term $t \cdot |E|/p$ over the offset array of the graph CSR. As a result of this work division, each thread is then responsible for a non-overlapping set of vertices and computes the number of comparisons required by each vertex. The results are then stored in an array of size $O(|V|)$. Note that computing the expected number of comparisons needed by the algorithm requires the same number of the steps for both the edge-based and vertex-based approaches, with the key difference in the size of the array used. This is followed by a prefix sum over the $Work$ array of size $O(|V|)$. In the final step, a binary search is employed by each core to compute the partition points of the workload. As before, the vertices will be divided among the threads in such a way that their adjacency intersections will not be split.

3.3 Complexity analysis

In the edge-based approach, the amount of work per edge is maintained in an array of size $O(|E|)$. Hence, the space complexity of this approach is $O(|E|)$. For the vertex-based approach the space complexity is $O(|V|)$ due to the fact that work estimations are stored vertex by vertex. The time complexity (per thread) of the load balancing stage in both methods is decomposed as described in Table 2.

The work complexity is the time complexity multiplied by a factor of p cores. Overall, the work complexity is of $O(p \cdot log(|V|) + |E| + p \cdot log(|E|) + |E|)$ for the edge-based approach.

In the case of the vertex-based method the complexity is $O(p \cdot log(|V|) + p \cdot log(|E|) + |V| + |E|)$.

[2]This can occur when the ratio between the average vertex degree and number of cores is considerably small.

3.4 Vertex Cover Optimization

In this subsection we briefly discuss how our load balancing technique can be adapted to the vertex cover optimization presented in [13]. This optimization involves computing a vertex cover for the graph and doing the adjacency list intersection only when both vertices of the edge are in the vertex cover. They show that finding the vertex cover takes a small fraction of the total execution and that the requirement that both vertices of an edge be in the vertex cover can reduce the number of list intersections and number of comparisons. This optimization avoids counting the same triangle multiple times. Their optimization can be applied in addition to the lexicographical sorting which reduces the number of times triangles are counted by a factor of two. To adapt the vertex cover to our algorithm, two modifications are required: 1) Parallel computation of the vertex cover $\hat{V}$ and 2) apply the load balancing techniques discussed in this paper to the vertex cover, $\hat{V}$, instead of the entire vertex set V. Making these modifications allows creating a load balanced algorithm which avoids duplicate triangle counting.

3.5 Summary

We have shown two methods that load balance clustering coefficients. These approaches tradeoff accuracy for space complexity. For these methods to be considered asymptotically optimal, the load balancing phase must have a lower time complexity than the actual clustering coefficients computation from (4). For many real graphs, including sparse graphs, this will be the case as:

$$O(p \cdot log(|V|) + |E| + p \cdot log(|E|) + |E|) < O(|V| \cdot d_{max}^2) \quad (6)$$

for the edge-based case and

$$O(p \cdot log(|V|) + p \cdot log(|E|) + |V| + |E|) < O(|V| \cdot d_{max}^2) \quad (7)$$

for the vertex-based case.

As a result, both approaches will offer better performance and core-scaling. We discuss the overhead of this approach in Section 4 with respect to real graphs. Note that while the work complexity of clustering coefficients has not changed, the actual time complexity per core changes from $O(Work/p)$ to $\Theta(Work/p)$.

4. RESULTS

In this section, we present the experimental performance results for both our new parallel load balanced algorithms. In our tests, we use a 4-socket 40 physical core multicore system made up of the Intel Xeon E7-8870 processor. Each core runs at a 2.40 GHz frequency and has 30 MB of L3 cache per processor. The system has 256 GB of DDR3 DRAM. We test our algorithms over a subset of graphs from the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [1]. The graph set used in the tests can be found in Table 3.

We compare the performance of our algorithm with a straightforward parallel algorithm. For the straightforward algorithm, the vertices in the outer loop of Algorithm 1 are evenly split among the cores. The performance of the algorithms is dependent on the properties of the input graph. We distinguish two key characteristics that may affect substantially the scalability of the different methods such that the straightforward algorithm is likely to outperform our algorithms:

1) *Sparsity* - while most of the graphs are considered to be sparse, the road networks are especially sparse where $E \approx V$. The fact that such networks consist of a single connect component implies that the vertices have few adjacencies, which in turns means that the intersection stage will be considerably short. As such our algorithms may potentially introduce overhead for such networks.

2) *Uniform degree distribution* - for graphs in which the degree distribution is uniform and most vertices have the same number of neighbors the workload is inherently balanced as each vertex will require an equal number of comparisons. .

Table 2: Time complexities of both approaches

Stage	Edge-based approach	Vertex-based approach
Binary search in the offset array	$O(log(\|V\|))$	$O(log(\|V\|))$
Element-wise workload estimation	$O(\|E\|/p)$	$O(\|E\|/p)$
Workload prefix sum	$O(\|E\|/p + log(p))$	$O(\|V\|/p + log(p))$
Binary search in the workload prefix sum array	$O(log(\|E\|))$	$O(log(\|V\|))$

Table 3: Graphs from the 10th DIMACS Implementation Challenge used in our experiments with the clustering coefficients computation runtimes for the different algorithms. Labels: SF - Straightforward (40 threads), V-B: Vertex-Based (40 threads), and E-B: Edge-Based (40 threads).

Name	Network Type	$\|V\|$	$\|E\|$	Serial	SF	V-B	E-B
audikw1	Matrix	943k	38.35M	30.89	2.52	0.96	1.05
cage15	Matrix	5.15M	47.02M	18.98	1.10	0.70	0.76
ldoor	Matrix	952k	22.78M	7.94	0.26	0.25	0.28
astro-ph	Clustering	16k	121k	0.09	0.006	0.003	0.003
caidaRouterLevel	Clustering	192k	609k	0.39	0.07	0.01	0.01
cond-mat-2005	Clustering	40k	175k	0.10	0.009	0.003	0.004
in-2004	Clustering	1.38M	13.59M	32.67	5.58	1.19	1.21
coAuthorsCiteseer	Collaboration	227k	814k	0.26	0.02	0.01	0.01
coPapersCiteseer	Collaboration	434k	16M	21.37	3.64	0.86	0.88
coPapersDBLP	Collaboration	540k	15.24M	15.26	1.44	0.68	0.72
luxembourg	Road	114k	119k	0.01	0.0002	0.0005	0.0008
belgium	Road	1.44M	1.55M	0.14	0.005	0.007	0.008
road_central	Road	14.82M	16.93M	3.84	0.19	0.26	0.27
road_usa	Road	23.95M	28.85M	3.47	0.13	0.20	0.22
preferAttachment	Clustering	100k	499k	0.26	0.08	0.009	0.01
smallworld	Clustering	100k	499k	0.14	0.004	0.004	0.004
RMAT-18	Random	262k	10.58M	63.78	2.70	2.01	2.08
RMAT-20	Random	1.05M	44.62M	236.28	8.14	7.1	7.38

4.1 Scaling

We define the ratio of computed comparisons as the fraction of the minimum number of comparisons performed by a thread over the maximum number of comparisons. Fig. 2 depicts this workload imbalance for the three algorithms given a 40 core partition. In Fig. 3 we show this ratio for a subset of networks as a function of the number of threads. Our results show that the straightforward algorithm offers a balanced partitioning for the highly sparse networks with uniform degree distribution. Note that for all the networks, the straightforward approach has both the upper and lower bounds for work distribution. This is true for all the networks we tested. For some networks, including $caidaRouterLevel$, the ratio between the minimal and maximal workload can be as high as 100 times.

In contrast to the straightforward partitioning, our edge-based approach delivers a near equal number of comparisons to each core for all the networks. This fact can be observed in Fig. 2 (the ratio chart), where the bar for the edge-based method is approximately 1 for all the networks, which is the ideal scenario. Our observation can be reinforced by the data displayed in Fig. 3, as it is almost impossible to differentiate the two curves for minimal and maximal number of comparisons for the edge-based method. The vertex-based approach delivers mixed results. In some cases the partitioning overlaps with that of the edge-based approach. In some cases it differs by $10\% - 20\%$. This is due to some vertices being more computationally demanding and that these vertices are not split among several cores.

Results show that the partitioning of the vertex-based approach is not as accurate as that of the edge-based method. Despite this the vertex-based approach offers better performance, as its load balancing stage is slightly less computationally demanding.

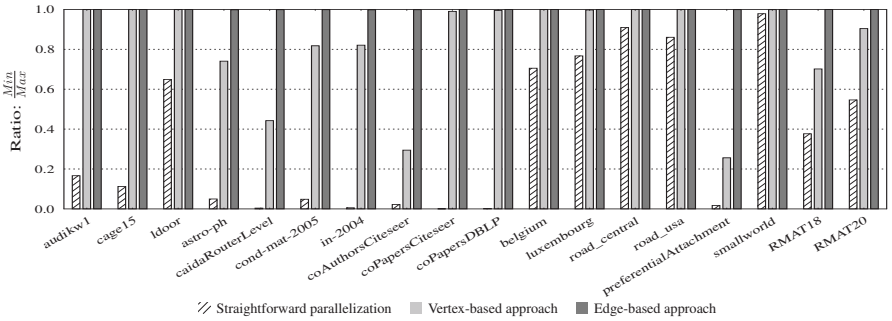
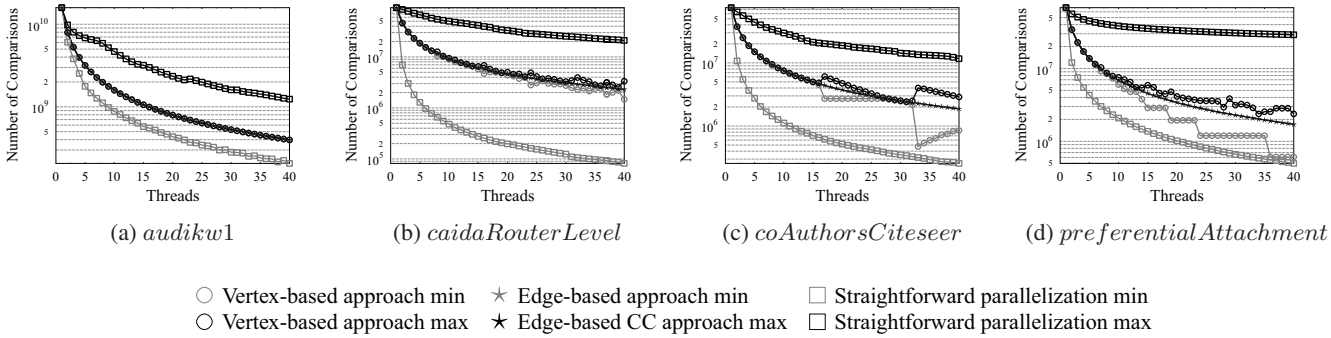


Figure 2: The ordinate is the ratio between the thread with the least amount of work with the thread with the most amount of work based on the number comparisons required by the thread in the adjacency list intersection. The abscissa are the graphs used. Note that for all the graphs the edge-based approach achieves almost perfect partitioning.



(a) $audikw1$ (b) $caidaRouterLevel$ (c) $coAuthorsCiteseer$ (d) $preferentialAttachment$

Figure 3: The abscissa is the number of threads used. The ordinate is the number of comparisons required for a specific thread count. Two curves are shown for each of the algorithms: for threads that receive the most and least number of comparisons. The straightforward partitioning is the lower and upper for all the figures while the minimum and the maximum overlap in the case of the edge-based approach.

The number of comparisons each processor receives can be also relevant to estimate an upper bound on the speedup a given parallel clustering coefficients calculation can attain. Given a work distribution, the maximum speedup obtained by parallel computation can be expressed as in expression (1). In the next subsection, we show how such theoretical speedup displays a correlation with the actual speedup attained for the different networks in the set.

4.2 Speedup Analysis

Fig. 4 depicts the speedup obtained for the three algorithms when using 40 cores. Further details of the strong scaling speedups are shown in Fig. 5, as a function of the number of threads.

If we consider the small world graphs, both of our methods show better scalability for 40 cores, which represents an improvement over the straightforward algorithm of $1.5X - 7.5X$. For road network graphs, the straightforward algorithm outperforms both our methods. This is due to the fact that road networks are very sparse, $E \approx V$. As a result, the computation of the intersection represents a smaller fraction of the overall runtime, which includes the load-balancing. In addition, road networks feature a substantially uniform degree distribution. Hence, a straightforward division of the work yields a load-balanced computation. The same behavior can be observed in other networks that feature uniform degree distributions, such as the $smallworld$ network. A clear relationship

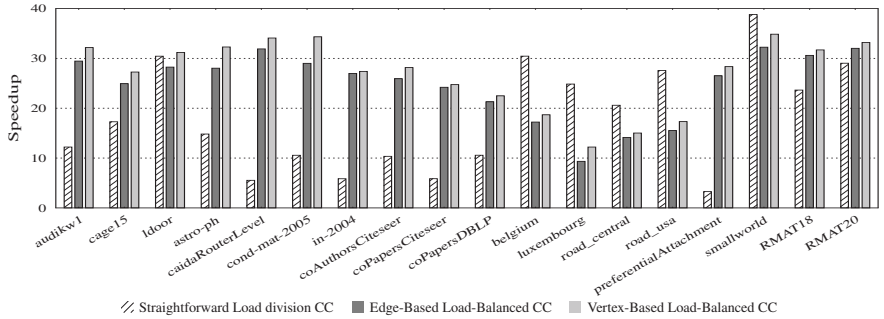


Figure 4: Speedup obtained with 40 cores for the different algorithms

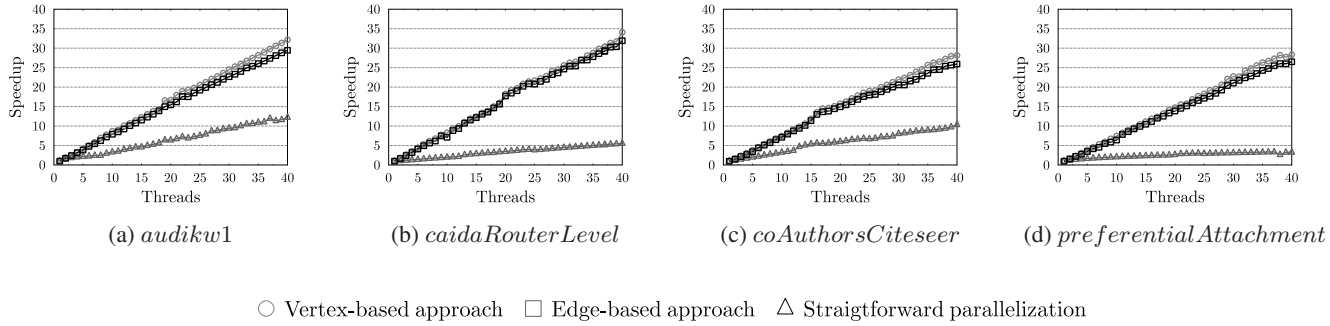


Figure 5: The ordinate for all the subfigures is the speedup as a function of the number of threads (the abscissa).

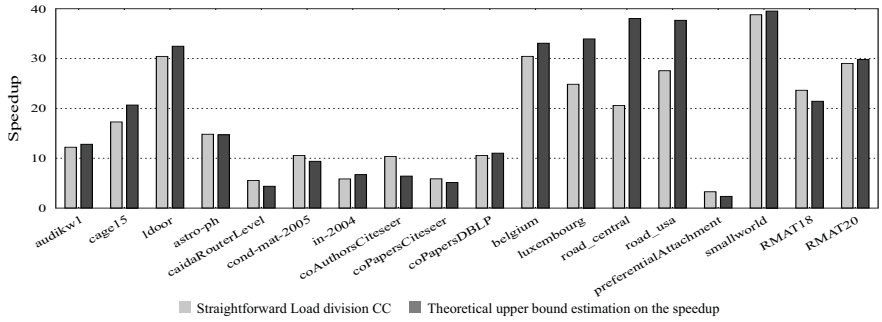


Figure 6: Speedup obtained for 40 cores using straightforward division algorithm in comparison with the estimated speedup.

can be observed between the thread with the most work and the actual speedup. This is not surprising, given the fact that this thread is the execution bottleneck.

Fig. 6 shows a comparison between the actual speedups obtained for the different networks when using the straightforward approach and an estimated maximum speedup obtained using expression (1). Overall, our work estimations provide a fairly precise indicator on the behavior of the algorithm for most of the networks.

Further insights on the overhead of the work estimation phase are depicted in Fig. 7, which shows the ratio of the time spent computing the clustering coefficients out of the total time spent (including the load balancing) for both our methods on the 40 cores for all the networks we used. The overhead of our load balancing techniques represents between 1% - 20% of the overall runtime, except for the road network graphs. For the road networks, our overhead is indeed significant. This is mostly due to the fact that very little work is done in the adjacency list intersection stage meaning that the overhead introduced by our techniques plays a more significant role in the overall time.

Note, if the load-balancing stage is not taken into account in the total execution time, but rather just the clustering coefficients execution time, our new algorithms always outperform the straightforward algorithm for all thread counts due to load balancing.

Despite the fact that the edge-based version provides a more balanced work distribution than the vertex-based method, these differences do not translate into a better performance. This is due to the higher computational complexity of the work estimation phase of our edge-based method and synchronization. For all the net-

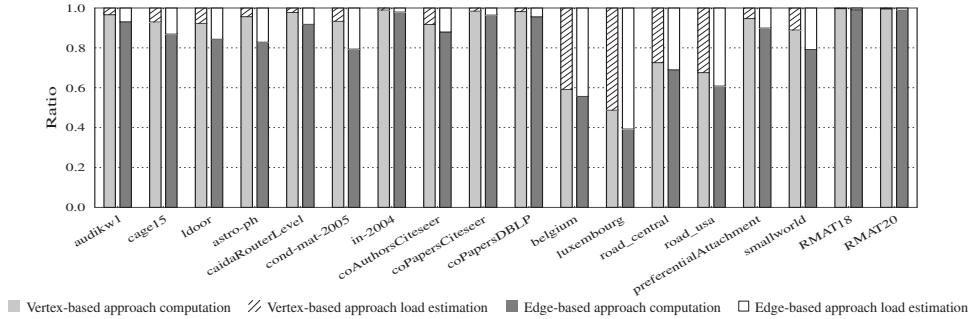


Figure 7: Percentage of time spent in both the load balancing phase vs. the list intersection phase for both approaches.

works, the edge-based approach introduces more overhead than the vertex-based approach. This is caused by the increased memory and computational requirements of this method. Recall that the edge-based approach uses an $O(E)$ array to store the expected amount of work. This array is then used for the prefix sum process for a total of $O(E)$ operations (whereas the vertex based requires only $O(V)$ operations). Further, this prefix array will be reloaded into the cache for each of the accesses.

4.3 Summary

This section presented timing results, scaling results, and workload partitioning for the straightforward parallel implementation, our edge and vertex-based approach using real world graphs from the DIMACS Challenge [1]. We showed that both our load balancing techniques scaled up to 40 cores and can continue to scale to a significantly larger number of cores, whereas the scalability of straightforward algorithm is limited for many types of networks. In some cases, our approaches show an improvement of a factor of as high as $5.5X - 7.5X$ over the straightforward algorithm. The overhead introduced by our algorithm is discussed.

5. CONCLUSIONS

Due to highly skewed vertex degree distributions, computing clustering coefficients on social networks presents big load balancing challenges. In this paper we presented two parallel methods for computing exact clustering coefficients. By using workload estimation, we achieve effective load-balanced computation for multiple graph topologies. For both of our methods, we present a discussion on the tradeoffs between achieving perfect work distribution and the complexity it requires. In practice, employing an approximate load balancing scheme with a moderate computational cost allows achieving an overall speedup of $25X - 35X$ over the sequential algorithm for most of the graphs. This represents an improvement of $3X - 7.5X$ for real graphs and $1.5X - 4X$ for random graphs over using straightforward parallel approaches. Overall, load balancing is a key element to take into account when leveraging the parallel computing power in graph applications.

6. REFERENCES

[1] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering*, volume 588. American Mathematical Society, 2013.

[2] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *ACM-SIAM Symposium on Discrete algorithms*, SODA '02, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[3] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[4] A.-L. Barabási and Z. N. Oltvai. Network Biology: Understanding the Cell's Functional Organization. *Nature Reviews Genetics*, 5(2):101–113, 2004.

[5] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler. Implementing a portable multi-threaded graph library: The MTGL on Qthreads. In *IEEE International Symposium on Parallel & Distributed Processing, IPDPS*, pages 1–8. IEEE, 2009.

[6] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs. In *ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 16–24. ACM, 2008.

[7] G. E. Blelloch. Prefix sums and their applications. 1990.

[8] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph Structure in the Web. *Computer Networks*, 33(1):309–320, 2000.

[9] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting Triangles in Data Streams. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 253–262. ACM, 2006.

[10] D. Ediger, K. Jiang, J. Riedy, and D. Bader. GraphCT: Multithreaded Algorithms for Massive Graph Analysis. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2012.

[11] D. Ediger, K. Jiang, J. Riedy, and D. A. Bader. Massive Streaming Data Analytics: A Case Study with Clustering Coefficients. In *International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[12] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of The Internet Topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[13] O. Green and D. A. Bader. Faster Clustering Coefficients Using Vertex Covers. In *5th ASE/IEEE International Conference on Social Computing*, SocialCom, 2013.

[14] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[15] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[16] A. Leist, K. Hawick, D. Playne, and N. S. Albany. GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs. In *International Conference on Scientific Computing (CSC'11)*, 2011.

[17] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[18] S. Milgram. The Small World Problem. *Psychology Today*, 2(1):60–67, 1967.

[19] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - parallel merging made simple. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1611–1618, 2012.

[20] T. Schank and D. Wagner. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[21] Y. Shavitt and E. Shir. DIMES: Let the Internet Measure Itself. *ACM SIGCOMM Computer Communication Review*, 35(5):71–74, 2005.

[22] S. H. Strogatz. Exploring Complex Networks. *Nature*, 410(6825):268–276, 2001.

[23] D. J. Watts and S. H. Strogatz. Collective Dynamics of "Small-World" Networks. *Nature*, 393(6684):440–442, 1998.