Rectangle Counting in Large Bipartite Graphs

Jia Wang, Ada Wai-Chee Fu, James Cheng
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{jwang,adafu,jcheng}@cse.cuhk.edu.hk

***Abstract*—Rectangles are the smallest cycles (i.e., cycles of length 4) and most elementary sub-structures in a bipartite graph. Similar to triangle counting in uni-partite graphs, rectangle counting has many important applications where data is modeled as bipartite graphs. However, efficient algorithms for rectangle counting are lacking. We propose three different types of algorithms to cope with different data volumes and the availability of computing resources. We verified the efficiency of our algorithms with experiments on both large real-world and synthetic bipartite graphs.**

***Keywords*-bipartite graphs; rectangle counting;**

I. Introduction

A bipartite graph models the relationship between two sets of entities in many applications. Examples of such bipartite relationship include authors and papers in scientific collaboration, movies and reviews in recommender sites, queries and URLs in query logs in Web search applications, customers and purchased items in market baskets, etc.

In this paper, we propose to study the fundamental substructures in a bipartite graph. In uni-partite graphs, **triangles** (i.e., cycles of length 3) are considered as the building blocks of a graph [1], [2], [3], which play a central role in graph and network analysis. In a bipartite graph, triangles do not exist and the corresponding patterns are the **rectangles**, i.e., *the smallest cycles or cycles of length 4*.

Intuitively, a rectangle can be interpreted as: the relationship between two objects at one side of the bipartite graph is reinforced by a double connection to two objects at the other side of the bipartite graph. From another angle, a rectangle implies that two objects having the same connecting point are likely to have another common connecting point. For example, in an author-paper network, a rectangle indicates that two authors collaborate at least twice in publishing, and they are more likely to collaborate again with each other than with those whom they have never collaborated.

We study the problem of **rectangle counting**, that is, *counting the number of rectangles in a bipartite graph*. For uni-partite graphs, triangle counting is an extensively studied problem [4], [5], [6], [7], [8], [1], [2], [9], [10] and applied in the analysis of various networks and graphs [11], [3]. Since rectangles are the counterpart of triangles in bipartite graphs, rectangle counting can also be applied to study bipartite graphs in similar ways as triangle counting for uni-partite graphs. In particular, rectangle counting lies at the heart of the computation of important network analysis metrics for bipartite graphs [12], [13], [14], [15] such as (global and local) clustering coefficients, rectangle-based connectivity, and the estimation of larger cycles by counting 4-cycles.

Although there are many applications and important rectangle-based measures/concpets have been proposed [12], [13], [14], [15], we are not aware of any efficient algorithms for rectangle counting. Since rectangles are different in structure from triangles and bipartite graphs have features different from uni-partite graphs, algorithms for triangle counting in uni-partite graphs cannot be applied for rectangle counting in bipartite graphs.

In this paper, we propose three efficient algorithms for rectangle counting. The first algorithm is an in-memory algorithm. The algorithm is especially fast for computation in bipartite graphs of small to moderate size.

The second algorithm addresses a limitation of the first algorithm, that is, the in-memory algorithm is inefficient to handle graphs that are too large to fit in main memory. Many real-world bipartite networks have grown very large and are still growing at a steady rate. When the input graph cannot fit in main memory, the pattern of edge accesses of the in-memory algorithm incurs high I/O cost due to random disk access. Thus, we develop an I/O-efficient algorithm to reduce the I/O cost for rectangle counting in large graphs.

Our third algorithm considers parallel rectangle counting. For processing massive volume of data, MapReduce has become a standard platform for providing large-scale distributed computation. We adopt the MapReduce framework for rectangle counting, and prove that the total amount of work done by all machines to run the algorithm is the same as that by the in-memory algorithm. In addition, we also propose a partition-based parallel algorithm and found that the partition-based algorithm is significantly more efficient and scalable than the MapReduce algorithm. Our result shows that MapReduce may not be most suitable for tasks such as rectangle counting.

The three algorithms cope with graphs of different sizes, and also consider the availability of computing resources (both memory and CPU resource). Our algorithms are simple in design and shown to be very efficient. In particular, our algorithms have $O(\sum(deg(v))^2)$ CPU time complexity, which even in the worst case is much lower than the $O(n^4)$ bound on the number of rectangles in a bipartite graph, where $deg(v)$ is the degree of a vertex v and n is the number of vertices at one side of the bipartite graph. Note that

the lowest known time complexity for triangle counting is $O(n^{2.376})$ [16], while the bound on the number of triangles is $O(n^3)$ for a graph with n vertices. The fastest practical algorithm takes $O(m^{1.5})$ [2], [9] for a graph with m edges, which is essentially an enumeration of all triangles and thus is the same as the bound on the number of triangles in the worst case. Our algorithms do not enumerate rectangles and hence have a much lower complexity than the $O(n^4)$ bound on the number of rectangles.

We evaluated our algorithms on both large real-world and synthetic bipartite graphs. Our results show that our in-memory algorithm is at least an order of magnitude faster than the existing small-cycle counting algorithm with the best known time complexity [16], with a significantly less memory consumption. The lack of efficient algorithms in practice shows the need for more efficient algorithms for rectangle counting, without which the existing proposals of bipartite graph analysis using clustering coefficients [12], [13], [14], [15] cannot be made possible.

For larger graphs, the experimental results show that our I/O-efficient algorithm can process rectangle counting efficiently with limited main memory. The MapReduce algorithm also shows reasonable speedup with growing computing capacity, although we show that our partition-based parallel algorithm is dramatically more efficient than the MapReduce algorithm. Finally, we present some interesting findings on the networks by studying clustering coefficients defined based on rectangles.

Outline. Section II gives the notations and problem definition. Sections III, IV, and V present the in-memory, I/O-efficient, and parallel algorithms, respectively. Section VI discusses clustering coefficient for bipartite graphs. Section VII reports the experimental results. Section VIII discusses related work and Section IX concludes the paper.

II. Problem Definition

Let $G = (V_G = (L_G \cup R_G), E_G)$ be an undirected, unweighted bipartite graph, where L_G and R_G are two disjoint sets of vertices of G and $E_G \subseteq \{\{u, v\} : u \in L_G, v \in R_G\}$ is the set of edges of G. An edge between u and v are denoted by either (u, v) or (v, u).

We assume that the graph is stored in its adjacency list representation (whether in main memory or on disk), where vertices are assigned with unique IDs.

We define the set of *adjacent vertices* of a vertex $v \in V_G$ as $adj(v) = \{u : (u, v) \in E_G\}$, and the *degree* of v in G as $deg(v) = |adj(v)|$. Given two distinct vertices $v, v' \in L_G$ (or $v, v' \in R_G$), we define the set of common adjacent vertices of v and v' as $adj(v, v') = \{u : (v, u) \in E_G, (v', u) \in E_G\}$.

We define a **rectangle** in G as follows. A rectangle is a *complete bipartite subgraph* consisting of the vertex set $\{\ell, \ell', r, r'\}$ such that $\ell, \ell' \in L_G$ and $r, r' \in R_G$. We denote the rectangle by $[\ell, r, \ell', r']$. The rectangle $[\ell, r, \ell', r']$ can also be seen as a *length-4 cycle* consisting of the following edges: $(\ell, r), (r, \ell'), (\ell', r'), (r', \ell)$.

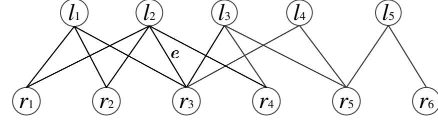

Figure 1. Rectangle counting

We denote by $\gamma(v)$ and $\gamma(\ell, r)$ the number of rectangles containing the vertex v and the edge (ℓ, r) in G, respectively. We also use $\sum \gamma(\ell)$ to denote $\sum_{\ell \in L_G} \gamma(\ell)$, and $\sum \gamma(r)$ to denote $\sum_{r \in R_G} \gamma(r)$. The number of rectangles in G, denoted by $\gamma(G)$, is given as follows

$$\gamma(G) = \frac{1}{2} \sum \gamma(\ell) = \frac{1}{2} \sum \gamma(r). \tag{1}$$

Equation 1 holds as every rectangle is counted twice by the two vertices at each side of the bipartite graph.

The following example illustrates the concepts.

Example 1: Figure 1 shows a bipartite graph G with $L_G = \{l_1, l_2, l_3, l_4, l_5\}$ and $R_G = \{r_1, r_2, r_3, r_4, r_5, r_6\}$. There are 5 rectangles in G, namely $[l_1, r_1, l_2, r_2]$, $[l_1, r_1, l_2, r_3]$, $[l_1, r_2, l_2, r_3]$, $[l_2, r_3, l_3, r_4]$, and $[l_3, r_3, l_4, r_5]$. Thus, $\gamma(G) = 5$. We can also count the number of rectangles containing any vertex or edge. For example, we have $\gamma(l_1) = 3$, $\gamma(l_2)=4$, $\gamma(r_1)=2$, and $\gamma(e)=3$ where $e=(l_2, r_3)$. □

Problem Definition. Given a bipartite graph $G = (V_G = (L_G \cup R_G), E_G)$, the problem of **rectangle counting** is to compute $\gamma(v)$ for each vertex $v \in L_G$ (or $v \in R_G$), $\gamma(e)$ for each edge $e \in E_G$, and $\gamma(G)$.

III. In-Memory Rectangle Counting

We first present an in-memory algorithm for rectangle counting in a bipartite graph, as shown in Algorithm 1.

Given a bipartite graph $G = (L_G \cup R_G, E_G)$, the algorithm sequentially processes each vertex $\ell \in L_G$ to compute $\gamma(\ell)$, along with $\gamma(e)$ for each $e = (\ell, r) \in E_G$, as follows.

Let $2hop(\ell)$ be the set of vertices that are *exactly* two hops (i.e., two edges) away from ℓ. Note that $2hop(\ell) \subseteq L_G$, since $\ell \in L_G$ and every path of length 2 from ℓ must end at some vertex $\ell' \in (L_G \backslash \ell)$.

The algorithm first computes $2hop(\ell)$ and $adj(\ell, \ell')$ for each $\ell' \in (L_G \backslash \ell)$ in Lines 5-10, which is by doing a two-hop depth-first traversal from ℓ. Then, $\gamma(\ell)$ and $\gamma(e)$, for each $e=(\ell, r) \in E_G$, are computed from $2hop(\ell)$ and $adj(\ell, \ell')$ in Lines 11-14. Finally, the algorithm also obtains $\gamma(G)$ from $\gamma(\ell)$ for all $\ell \in L_G$. The correctness and complexity of the algorithms are established by the following theorems.

Theorem 1: Given a bipartite graph $G = (L_G \cup R_G, E_G)$, Algorithm 1 correctly computes $\gamma(\ell)$ for each vertex $\ell \in L_G$, $\gamma(e)$ for each edge $e \in E_G$, and $\gamma(G)$.

Algorithm 1: In-Memory Rectangle Counting

Input : A bipartite graph $G = (V_G = (L_G \cup R_G), E_G)$
Output : $\gamma(\ell)$ for each $\ell \in L_G$, $\gamma(e)$ for each $e \in E_G$, and $\gamma(G)$

1 **begin**
2    $\gamma(\ell) \leftarrow 0$ for each $\ell \in L_G$;
3    $\gamma(e) \leftarrow 0$ for each $e \in E_G$;
4   **foreach** $\ell \in L_G$ **do**
5      $2hop(\ell) \leftarrow \emptyset$;
6      $adj(\ell, \ell') \leftarrow \emptyset$ for each $\ell' \in (L_G \backslash \ell)$;
7   **foreach** $r \in adj(\ell)$ **do**
8   **foreach** $\ell' \in adj(r) \backslash \ell$ **do**
9          $adj(\ell, \ell') \leftarrow adj(\ell, \ell') \cup \{r\}$;
10          $2hop(\ell) \leftarrow 2hop(\ell) \cup \{\ell'\}$;
11   **foreach** $\ell' \in 2hop(\ell)$ **do**
12        $\gamma(\ell) \leftarrow \gamma(\ell) + \binom{|adj(\ell,\ell')|}{2}$;
13   **foreach** $r \in adj(\ell, \ell')$ **do**
14          $\gamma(\ e{=}(\ell, r)\ ) \leftarrow \gamma(e) + |adj(\ell, \ell')| - 1$;
15    $\gamma(G) \leftarrow \frac{1}{2} \sum \gamma(\ell)$;

Theorem 2: Algorithm 1 uses $O(\sum(deg(r))^2)$ time and $O(|V_G| + |E_G|)$ memory space.

Proof: The proofs can be found in [17]. ■

We illustrate how Algorithm 1 works as follows.

Example 2: Given the graph in Figure 1, consider the processing of $\ell = l_2$ in Lines 4-14. Since $adj(l_2) = \{r_1, r_2, r_3, r_4\}$, we access $adj(r_1)$, $adj(r_2)$, $adj(r_3)$ and $adj(r_4)$ to compute the 2-hop neighbors of l_2 and obtain $2hop(l_2) = \{l_1, l_3, l_4\}$. We also obtain $adj(l_2, l_1) = \{r_1, r_2, r_3\}$, $adj(l_2, l_3) = \{r_3, r_4\}$, and $adj(l_2, l_4) = \{r_3\}$. In Line 12, we add $\binom{|adj(l_2,l_1)|}{2} = 3$ and $\binom{|adj(l_2,l_3)|}{2}$=1 to $\gamma(l_2)$, but 0 from $adj(l_2, l_4)$ since $|adj(l_2, l_4)|$=1. For the edges, e.g., $e = (l_2, r_3)$, since r_3 is in $adj(l_2, l_1)$, $adj(l_2, l_3)$, and $adj(l_2, l_4)$, Line 14 sums up each of their sizes minus 1, thus we have $\gamma(e) = (3-1)+(2-1)+(1-1) = 3$. □

IV. I/O-Efficient Rectangle Counting

We now discuss our second algorithm, an I/O-efficient algorithm, which handles the case when the input graph is too large to fit in main memory.

To avoid random access to the graph stored on disk, the algorithm first partitions L_G into a set of p disjoint vertex sets, $\mathcal{P}ar = \{P_1, P_2, \ldots, P_p\}$. Then, for each vertex set $P \in \mathcal{P}ar$, we construct the **neighborhood subgraph** of P, denoted by $NG(P)$, which is simply the subgraph consisting of all edges incident to vertices in P.

With the above formulation of $NG(P)$, the partitioning can be easily done by sequentially scanning G (in its adjacency-list representation) as follows. We first obtain P_1, which is the first $|P_1|$ vertices of L_G read from G such

Algorithm 2: I/O-Efficient Rectangle Counting

Input : A bipartite graph $G = (V_G = (L_G \cup R_G), E_G)$
Output : $\gamma(\ell)$ for each $\ell \in L_G$ and $\gamma(e)$ for each $e \in E_G$

1 **begin**
2   partition L_G into $\mathcal{P}ar = \{P_1, P_2, \ldots, P_p\}$;
3   **foreach** $P \in \mathcal{P}ar$ **do**
4   construct $H = NG(P)$;
5   call Algorithm 1 with H as input to compute
6        $\gamma(\ell, H)$ for each $\ell \in L_H$, and
7        $\gamma(e, H)$ for each $e \in E_H$;
8   **for each** $\ell \in L_H$: $\gamma_1(\ell) \leftarrow \gamma(\ell, H)$;
9   **for each** $e \in E_H$: $\gamma_1(e) \leftarrow \gamma(e, H)$;
10   **foreach** *pair* $P_i, P_j \in \mathcal{P}ar$*, where* $i < j$ **do**
11      $P \leftarrow P_i \cup P_j$;
12   construct $H = NG(P)$;
13   call Algorithm 1 with H as input to compute
14        $\gamma(\ell, H)$ for each $\ell \in L_H$, and
15        $\gamma(e, H)$ for each $e \in E_H$;
       // $\gamma_2(.)$ is initialized as 0 for first-time use
16   **for each** $\ell \in L_H$: $\gamma_2(\ell) \leftarrow \gamma_2(\ell) + \gamma(\ell, H) - \gamma_1(\ell)$;
17   **for each** $e \in E_H$: $\gamma_2(e) \leftarrow \gamma_2(e) + \gamma(e, H) - \gamma_1(e)$;
18   **for each** $\ell \in L_G$: $\gamma(\ell) \leftarrow \gamma_1(\ell) + \gamma_2(\ell)$;
19   **for each** $e \in E_G$: $\gamma(e) \leftarrow \gamma_1(e) + \gamma_2(e)$;

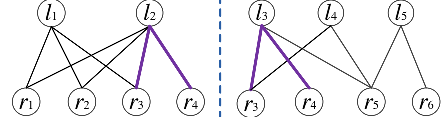

Figure 2. Two types of rectangles

that $NG(P_1)$ fills up half of the available memory[1]. We can simply construct $NG(P_1)$ from the adjacency lists of the vertices in P_1. After we process $NG(P_1)$, we move on to obtain P_2 and construct $NG(P_2)$ in the same way, and so on until we process all vertices in L_G.

With respect to the partition $\mathcal{P}ar$, we can classify the rectangles in G into two types as follows. Since each rectangle has two vertices in L_G (let us call them **left-vertices**), we have: (1) **Type-1 rectangles** with both left-vertices in the same P, for some $P \in \mathcal{P}ar$; or (2) **Type-2 rectangles** with the two left-vertices in two different P_i and P_j, for some $P_i, P_j \in \mathcal{P}ar$. We use $\gamma_1(.)$ and $\gamma_2(.)$ as the count of the two types of rectangles, respectively.

Consider the graph in Figure 1, we show how it is partitioned into two parts in Figure 2. Note that r_3 and r_4 are repeated in the two parts since they are in the adjacency lists of l_2 and also l_3. Rectangles falling into any single part, including $[l_1, r_1, l_2, r_2]$, $[l_1, r_1, l_2, r_3]$, $[l_1, r_2, l_2, r_3]$, $[l_3, r_3, l_4, r_5]$, are Type-1 rectangles. There is only one Type-2 rectangle in this case, namely $[l_2, r_3, l_3, r_4]$.

With the above set-up, Algorithm 2 mainly consists of two for-loops. The first for-loop (Lines 3-9) counts Type-

[1] We use only half of the memory for $NG(P_1)$ because in Algorithm 2 we merge $NG(P_i)$ and $NG(P_j)$ for $P_i, P_j \in \mathcal{P}ar$.

1 rectangles, which calls Algorithm 1 with $H = NG(P)$ as input for each $P \in \mathcal{P}ar$. Let $\gamma(\ell, H)$ and $\gamma(e, H)$ be the number of rectangles in H containing $\ell \in L_H$ and $e \in E_H$, respectively. Then, we simply have $\gamma_1(\ell) = \gamma(\ell, H)$ for each $\ell \in L_H$ and $\gamma_1(e) = \gamma(e, H)$ for each $e \in E_H$. The second for-loop (Lines 10-17) counts Type-2 rectangles, which calls Algorithm 1 with $NG(P_i \cup P_j)$ as input for each pair $P_i, P_j \in \mathcal{P}ar$, where $i < j$, since now the two left-vertices of each rectangle are in two different P_i and P_j. Then, we compute $\gamma_2(.)$ correspondingly. Finally in Lines 18-19, we compute $\gamma(\ell)$ for each $\ell \in L_G$ and $\gamma(e)$ for each $e \in E_G$ from $\gamma_1(.)$ and $\gamma_2(.)$. The following theorem establishes the correctness of the counting.

Theorem 3: Given a bipartite graph $G = (L_G \cup R_G, E_G)$, Algorithm 2 correctly computes $\gamma(\ell)$ for each vertex $\ell \in L_G$ and $\gamma(e)$ for each edge $e \in E_G$.

Proof: The proof can be found in [17]. ■

We now analyze the complexity of Algorithm 2. We use the following standard I/O complexity notations in the analysis: M is the main memory size, B is the disk block size, $scan(N) = \Theta(N/B)$ I/Os, where $1 \ll B \leq M/2$ and N is the amount of data being read/written from/to disk.

Theorem 4: Algorithm 2 uses $O(\frac{|E_G|}{M} scan(|V_G| + |E_G|))$ I/Os and $O(\sum (deg(r))^2)$ CPU time.

Proof: The proof can be found in [17]. ■

V. Parallel Rectangle Counting

We discuss two parallel algorithms, a partition-based algorithm and a MapReduce algorithm.

A. Partition-Based Algorithm

Assume that there are p available machines. A master machine takes the input graph and distributes data and tasks to other machines. We describe the algorithm as follows.

We randomly partition L_G into a set of p disjoint vertex sets, $\{L_1, \ldots, L_p\}$. Let $R_i = \bigcup_{\ell \in L_i} adj(\ell)$. Then, we construct the neighborhood subgraph of R_i, $NG(R_i)$, i.e., the subgraph consisting of all edges incident to vertices in R_i. We distribute $NG(R_i)$ to machine i, for $1 \leq i \leq p$, and machine i applies Algorithm 1 (with $NG(R_i)$ as input) to count rectangles that involve vertices in L_i. We name this parallel algorithm as **PAR-rect**. The execution at machine i is shown in Algorithm 3.

The following theorem (proof is given in [17]) shows that the total amount of work done by PAR-rect is the same as that by the in-memory algorithm, i.e., Algorithm 1.

Theorem 5: Given a bipartite graph $G = (L_G \cup R_G, E_G)$, Algorithm PAR-rect correctly computes $\gamma(\ell)$ for each vertex $\ell \in L_G$ and $\gamma(e)$ for each edge $e \in E_G$. The total amount of work performed by all machines to run parallel rectangle counting is $O(\sum (deg(r))^2)$.

Algorithm 3: Execution of **PAR-rect** at Machine i

Input : L_i and $NG(R_i)$
Output : $\gamma(\ell)$ for each $\ell \in L_i$, $\gamma(\ell, r)$ for each $\ell \in L_i$ and $r \in R_i$

```
1 begin
2     run Algorithm 1 with NG(R_i) as input, with the
      exception that in Line 4 substitute ℓ ∈ L_i for ℓ ∈ L_G,
      and ignore Line 15;
```

Having the same total workload as the in-memory algorithm does not imply good parallelization, since the performance also depends on the even distribution of workload, as a bottleneck in one machine can seriously affect the overall runtime. We now analyze the workload balancing performance achieved by PAR-rect. From Algorithm 3, the total CPU workload is made up of the workload of each vertex in L_G. The CPU workload from each vertex $\ell \in L_G$ is given by $W(\ell) = \sum_{r \in adj(\ell)} deg(r)$.

If L_i is assigned to machine i, then the total CPU workload for machine i is given by $W_i = \sum_{\ell \in L_i} W(\ell)$. Let $L_G = \{\ell_1, ..., \ell_n\}$. Note that each machine gets $\frac{n}{p}$ vertices, i.e. $|L_i| = \frac{n}{p}$. As the assignment of vertices in L_i is random, we consider the selection of L_i, for $1 \leq i \leq p$, a random sampling from L_G. Thus, L_G is our sample space of size n, and each sample ℓ has a weight of $W(\ell)$. Let $\mu = \frac{1}{n} \sum_{\ell \in L_G} W(\ell)$ be the mean workload per vertex (sample), and $\sigma^2 = \frac{1}{n} \sum_{\ell \in L_G} (W(\ell) - \mu)^2$ be the variance. We achieve a minimum CPU runtime (or the longest running time of any machine) C_{opt} for PAR-rect if workload is evenly distributed for the p machines, and thus the optimal time is given by $C_{opt} = \frac{n\mu}{p}$.

The sample average at machine i is given by $\frac{pW_i}{n}$. When the sample size is large enough, the central-limit theorem says that the value of $\sqrt{n}(\frac{pW_i}{n} - \mu)$ can be approximated by the normal distribution with mean 0 and variance σ^2. From this we can derive the following probabilistic guarantee of small deviation of the actual machine workload from the ideal workload.

Theorem 6: Given p machines running PAR-rect on a bipartite graph $G = ((L_G, R_G), E_G)$, where $|L_G| = n$, then for $1 \leq i \leq p$ and $0 < \epsilon < 1$,

$$Prob(W_i \leq (1+\epsilon)C_{opt}) \approx \frac{1}{2}\left(1 + erf\left(\frac{\epsilon\mu\sqrt{n/p}}{\sigma\sqrt{2}}\right)\right),$$

where erf is the error function.

The value of the error function in the above is high (close to 1 for the datasets we tested with $4 \leq p \leq 16$ and small values of ϵ), which implies that the actual workload at each machine i is close to the ideal workload with high probability.

B. MapReduce Rectangle Counting

MapReduce is popularly used to process large datasets. We present a MapReduce algorithm, called **MR-rect**, for

rectangle counting. MR-rect consists of two rounds of Map and Reduce, denoted by Map 1, Reduce 1, and Map 2, and Reduce 2, as follows. First, for each $r \in R_G$, Map 1 outputs $\langle(\ell, \ell'); r\rangle$ for all pairs $\ell, \ell' \in adj(r)$, where $\ell < \ell'$. In this way, Reduce 1 collects $adj(\ell, \ell')$ for each unique pair $\ell, \ell' \in L_G$, where ℓ and ℓ' are exactly 2 hops away and $\ell < \ell'$. Then, Reduce 1 simply outputs $\binom{|adj(\ell,\ell')|}{2}$ for each $\ell \in L_G$ and $(|adj(\ell, \ell')| - 1)$ for each edge $(\ell, r) \in E_G$, which are then collected and summed up by Reduce 2 to obtain $\gamma(\ell)$ and $\gamma(e)$ for each $\ell \in L_G$ and $e \in E_G$. Note that Map 2 is an identity mapper which does nothing.

The following theorems (proofs are given in [17]) show the correctness and that the total amount of work done by MR-rect is the same as that by Algorithm 1.

Theorem 7: Given a bipartite graph $G = (L_G \cup R_G, E_G)$, Algorithm MR-rect correctly computes $\gamma(\ell)$ for each vertex $\ell \in L_G$ and $\gamma(e)$ for each edge $e \in E_G$.

Theorem 8: The total amount of work performed by all machines to run Algorithm MR-rect is $O(\sum(deg(r))^2)$.

VI. Clustering Coefficients in Bipartite Graphs

Clustering coefficient is an important measure for network analysis. It measures how likely vertices in a graph tend to cluster together. For example, Figure 3(a) shows a loosely-connected bipartite network, while Figure 3(b) shows a tightly-knit network. Thus, we expect that Figure 3(b) has much higher clustering coefficient, both globally as a graph and locally at individual vertices, than Figure 3(a).

Both global and local clustering coefficients are well-known measures for analyzing uni-partite graphs (based on triangles) [11], [3]. A corresponding (local) clustering coefficient for bipartite graphs that was proposed in [15] is given by Equation 2 below.

Definition 1 (Clustering Coefficients): Given a bipartite graph $G = (V_G, E_G)$, the ***(local) clustering coefficient*** of a vertex $v \in V_G$, denoted by $\mathcal{C}(v)$, is defined as follows.

$$\mathcal{C}(v) = \frac{\gamma(v)}{\text{number of potential rectangles containing } v}. \quad (2)$$

The ***(global) clustering coefficient*** of G, denoted by $\mathcal{C}(G)$, is defined as

$$\mathcal{C}(G) = \frac{1}{|V_G|} \sum_{v \in V_G} \mathcal{C}(v). \quad (3)$$

Intuitively, $\mathcal{C}(v)$ measures the degree to which v forms tightly-knit community structures in G, while $\mathcal{C}(G)$ gives an overall indication of the clustering in G.

Zhang et al. [15] define the denominator of Equation (2) as follows.

$$\mathcal{C}(v) = \frac{\gamma(v)}{((deg(v) - 1)\sum_{u \in adj(v)}(deg(u) - 1)) - \gamma(v)}. \quad (4)$$

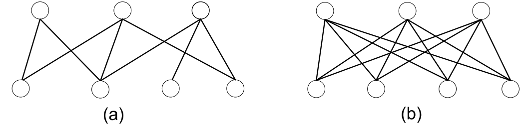

Figure 3. Loosely and tightly connected networks

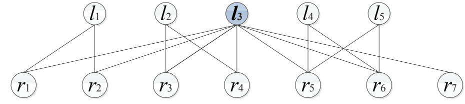

Figure 4. Two versions of local clustering coefficient

We give a different definition of the denominator of Equation (2) as follows.

$$\mathcal{C}(v) = \frac{\gamma(v)}{\binom{deg(v)}{2}|2hop(v)|}. \quad (5)$$

Intuitively, Equation (4) defines the number of potential rectangles containing v as the number of potential rectangles that can be formed with v by adding an extra edge to some length-3 path containing v, while Equation (5) considers the number of potential rectangles that can be formed out of v's neighbors (i.e., $adj(v)$) and those who share common neighbors with v (i.e., $2hop(v)$).

The main difference between the two equations is that Equation (5) gives a higher weight to a unified neighborhood structure centered at v; that is, Equation (5) tends to give a lower $\mathcal{C}(v)$ than Equation 4 if the neighborhood of v is divided into many small groups that are relatively isolated with each other. This is because for v to gain a high clustering coefficient by Equation (5), we require the subgraph induced by its neighbors and 2-hop neighbors are well connected *overall*, whereas Equation 4 is mainly determined by the number of shared neighbors of each pair of v's neighbors, putting little emphasis on the overall connection or community structures among these neighbors of v. We further illustrate using the following example.

Example 3: Consider the vertex l_3 in Figure 4, where its neighbors can be seen as divided into small groups, $\{r_1, r_2\}$, $\{r_3, r_4\}$, $\{r_5, r_6\}$, and $\{r_7\}$ (the grouping becomes clearer if we remove all edges incident to l_3). We have $\mathcal{C}(l_3) = 1/11$ by Equation (4), while Equation (5) gives a much lower $\mathcal{C}(l_3) = 1/21$ due to the separated neighborhood of l_3. □

The above example suggests that Equation (4) and Equation (5) may be used together to analyze a bipartite network. For example, if the clustering coefficients of the vertices computed by both equations are similar, it implies that the neighborhood structures of the vertices within the graph are well connected overall. On the contrary, it implies some scattered small structures within the neighborhood of the vertices, where the degree of scattered-ness may be quantified by the difference between the two clustering coefficients obtained by Equations 4 and 5. At the same,

we can still analyze from the results of both equations how likely the graph and its vertices tend to cluster.

Finally, $\mathcal{C}(v)$ defined by both Equations (4) and (5) can be obtained (with negligible overhead) by our algorithms which compute $\gamma(v)$ and also $|2hop(v)|$ during the process.

VII. Experimental Results

We evaluate the performance of our algorithms, denoted by **IM-rect**, **IO-rect**, **PAR-rect** and **MR-rect**, which represent ***I**n-**M**emory* algorithm, ***I/O***-efficient algorithm, ***PAR**tition-based* parallel algorithm, and ***M**ap**R**educe* algorithm for **rect**-angle counting, respectively. The fastest algorithm to the best of our knowledge that counts rectangles is for counting small cycles in general graphs by fast **M**atrix **M**ultiplication [16], denoted by **MM-rect**.

We ran all the sequential algorithms on a machine with an Intel(R) Core i3-2100 3.10GHz CPU, 3GB RAM, and Ubuntu 11.10 OS. Algorithm MR-rect was ran using Hadoop (version 0.20.205) on an Amazon Elastic MapReduce cluster with up to 20 nodes, each of which has the computing capacity of a 1.0 GHz 2007 Xeon processor, 1.7GB RAM, and 160GB instance storage. Algorithm PAR-rect was implemented using Message Passing Interface (MPI) and PAR-rect was tested on a cluster with up to 16 nodes each having an Intel Core 2 Duo 2.80GHz CPU and 4GB RAM.

Table I
DATASET STATISTICS (1K=10^3, 1M=10^6)

	$\|L_G\|$	$\|R_G\|$	$\|E_G\|$	disk size	$\gamma(G)$
movie	10K	72K	10M	103MB	1.2×10^{12}
netflix	5K	479K	75M	625MB	8.4×10^{13}
flickr	320K	1.6M	113M	957MB	9.7×10^{8}
deli	533K	17M	140M	2.2GB	1.8×10^{10}
trans	1M	80M	800M	13.5GB	5.3×10^{11}

Datasets. We use the following five datasets. The `movie` dataset is from the movie recommender service movielens.umn.edu, consisting of users and movies as vertices while each edge representing a rating. The `netflix` (netflix.com) is a dataset with users and actors as vertices and edges indicate the user has rent a movie in which the actor has a leading role. The Delicious (`deli`) dataset is from the social bookmarking web service delicious.com, where vertices are users and resources, and each edge is a tag assignment. The `flickr` dataset is from the online photo service flickr.com, where vertices are users and tags, and an edge is formed when a user assigns a tag to an image. The transaction (`trans`) dataset is a synthetic dataset generated by the IBM Quest Market-Basket synthetic data generator, where vertices are items and transactions, and an edge shows that a transaction contains an item. Some statistics, including the total number of rectangles, of the above networks are summarized in Table I.

A. Performance of In-Memory Computation

We first compare our in-memory algorithm, IM-rect, with the existing algorithm for counting 4-cycles, MM-rect [16]. Our experiments on MM-rect ran out of memory for all the datasets we used due to the $O(|V_G|^2)$ space requirement for matrix multiplication. Thus, we extract a smaller subgraph from each of the datasets. We select vertices from both sides of the bipartite graphs so that the ratio of the number of vertices on the two sides roughly follows that in the original graphs. The number of vertices and edges of each subgraph, $subg$, are given in Table II.

Table II
SIZE OF SUBGRAPHS (1K=10^3, 1M=10^6)

	movie	netflix	flickr	deli
$\|L_{subg}\|$	1.0K	100	1.3K	250
$\|R_{subg}\|$	7.0M	7.9K	6.7K	7.8K
$\|E_{subg}\|$	2.5M	238K	23K	48K

Table III
RUNNING TIME (IN SEC) AND PEAK MEMORY CONSUMPTION (IN MB) OF IM-RECT AND MM-RECT

	movie	netflix	flickr	deli
MM-rect (time)	62	51	12	50
IM-rect (time)	2	<1	<1	<1
MM-rect (mem)	512	512	512	512
IM-rect (mem)	37	4	20	14

Table III shows that IM-rect is more than an order of magnitude faster than MM-rect for all datasets, and uses significantly less memory. The result thus verifies that our algorithm very efficient, in terms of both running time and memory consumption, in processing small graphs.

Although it may not be fair to compare with MM-rect since it is not specifically designed for rectangle counting, the lack of efficient algorithms does demonstrate the need for such algorithms, as otherwise existing studies on the analysis of bipartite graphs using clustering coefficients [12], [13], [14], [15] will not be able to be carried out in practice.

B. Performance with Limited Memory

In this experiment, we evaluate the performance of our method when main memory is limited. We vary the available memory size from 0.5GB to 3.0GB. We report the results in Table IV. Whenever the entire graph can fit in the available memory, IO-rect is essentially the same as IM-rect. For these cases, we show the running time in bold numbers.

We report the results for the three larger datasets only. For the other two smaller datasets, `movie` and `netflix`, they can fit in just 0.5GB of memory and the running time of IM-rect is 11 seconds and 64 seconds, respectively.

The result shows that the in-memory algorithm, IM-rect, ran out of memory for `flickr` when memory is limited to 1.5GB or smaller, and for `deli` when memory is 2.5GB or less. For the `trans` dataset, IM-rect ran out of memory for all available memory sizes. On the contrary, the I/O-efficient algorithm, IO-rect, is able to process all datasets even when memory is limited to only 0.5GB.

For the datasets `flickr` and `deli`, the running time of IO-rect is comparable to that of IM-rect even though IO-rect uses much less memory. The main reason for this result,

Table IV
RUNNING TIME (IN SEC) AND NUMBER OF GRAPH SCANS BY IO-RECT

	3.0	2.5	2.0	1.5	1.0	0.5
Time (flickr)	**372**	**372**	**372**	368	368	373
# of scans (flickr)	1	1	1	2	2	4
Time (deli)	**594**	444	444	444	452	417
# of scans (deli)	1	2	2	2	3	6
Time (trans)	4251	5715	6490	8583	12570	23544
# of scans (trans)	8	10	13	18	28	58

as well as the stable running time of IO-rect over different available memory sizes, is because the I/O time of IO-rect is only about 1% of the total running time. In other words, the CPU time dominates the total running time of IO-rect, which thus matches the running time of IM-rect.

The performance of IO-rect for processing trans does not follow the trend of that for processing flickr and deli, as the running time increases roughly linearly with the decrease in the available memory size, or with the increase in the number of scans taken by IO-rect. However, the significantly larger number of scans required for trans than the other two datasets actually reveals that much more processing involving disk I/O is required for trans.

Overall, the result of this experiment shows that our I/O-efficient algorithm can effectively eliminate the prohibitively high I/O cost due to random disk access. The result also demonstrates the efficiency of our algorithms in processing both large and small bipartite graphs for rectangle counting, even when main memory resource is limited.

C. *Performances of Parallel Algorithms*

We now evaluate the performance of the parallel algorithms. For MR-rect, we measure the efficiency gain by examining the increase in the number of workers ranging from 5 to 20, and observe the trend using the datasets movie and netflix. For PAR-rect, we record the time by varying the number of workers from 1 to 16. We report the results in Table V and Table VI.

Table V
RUNNING TIME (IN MIN) BY MR-RECT

	5	10	15	20
movie	208	108	88	85
netflix	951	379	358	188

Table VI
RUNNING TIME (IN SEC) BY PAR-RECT

	1	2	4	8	16
movie	12.7	7.4	4.7	3.4	2.8
netflix	68.2	41.4	28.1	20.8	20.2
flickr	496	265	139	80	49
deli	892	469	261	158	110

It is clearly shown that PAR-rect outperforms MR-rect dramatically (note that the time shown in Table V is in minutes while the time shown in Table VI is in seconds). The inferior efficiency of MR-rect could be most likely explained by the quadratic amount of data generated and transmitted

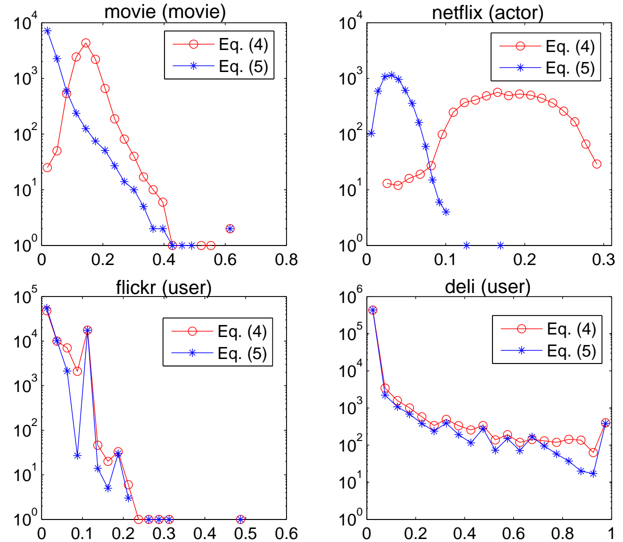

Figure 5. Distribution of clustering coefficient (x-axis: distribution interval, y-axis: frequency)

between machines, as well as the large sorting cost in the shuffle phase. Due to such limitations, we were not even able to obtain the results for the other larger datasets due to prolonged running time. On the contrary, PAR-rect can handle all datasets efficiently. The almost linear trend in the decrease of wall-clock time of PAR-rect with the increase in the number of workers also demonstrates the efficacy of our load-balancing mechanism.

The results also suggest that MapReduce may not be the first choice for solving graph problems of a similar nature as rectangle counting, when other alternative approaches such as I/O-efficient algorithms are possible. A recent work on I/O-efficient triangle counting [8] also reports much smaller running time in a single machine than the state-of-the-art MapReduce algorithm in 1636 machines [9].

D. *Clustering Coefficient Distribution*

We plot the distribution of local clustering coefficients, given by Equation 4 and Equation 5, for all the four real-world bipartite graphs, as shown in Figure 5. Note that a bipartite graph has vertices on two sides, the side we choose is given in parentheses next to the dataset name.

From Figure 5, the two different distributions of clustering coefficient show distinct patterns of the networks and reveal nontrivial information of the graphs. For deli and flickr, both distributions share a similar trend, which implies that the neighbors of most vertices are likely to intersect with each other on a significant portion of their neighborhood, i.e., vertices tend to form a tightly-knit community structure with their neighbors. On the contrary, for the netflix and movie networks, the two versions of distributions obviously deviate from each other. In particular, Equation 4 tends to give more vertices with a high

clustering coefficient than Equation 5 which only gives a high clustering coefficient to a smaller set of vertices with neighborhood being clustered as a whole.

In terms of highest value of clustering coefficient, we find that while it is only 0.3 for `netflix`, there exist vertices in `deli` that have clustering coefficient of nearly 1.0. Such information serves as an indicator of the global level of clustering in a bipartite graph. Furthermore, the small portion of vertices that have highest clustering coefficient in each network can be extracted for further analysis; for example, whether they form a core of the network.

Finally, one important finding of this experiment is that the use of Equation 4 and Equation 5 together can certainly reveal much more information than using either of them alone. And we remark that our algorithms can be applied to compute both distributions with negligible extra cost.

VIII. Related Work

Triangle counting in uni-partite graphs has been extensively studied [4], [5], [6], [7], [8], [1], [2], [9], [10]. However, due to the difference in structure (both in graph structure and in pattern structure), algorithms for triangle counting cannot be applied for rectangle counting. In uni-partite graphs, scalable algorithms for listing other important substructures such as maximal cliques and core subgraphs were studied in [18], [19], [20], [21], [22], [23].

For uni-partite graphs, counting length-k cycles, for any $k \leq 7$, takes $O(|V_G|^{\omega})$ time [16], where $\omega < 2.37$ is the exponent of matrix multiplication. For bipartite graphs, counting cycles of length g, $g+2$, and $g+4$ with girth $g > 6$ takes $O(g|V_G|^3)$ time also by method of matrix multiplication [24]. These algorithms, however, cannot scale to process even graphs of medium-size.

IX. Conclusions

We proposed the problem of rectangle counting in large bipartite graphs, and devised three types of algorithms to solve the problem to cope with different data volumes and available computing resources. Our experimental results showed that our in-memory algorithm is very efficient for processing small to medium size datasets, while our I/O-efficient algorithm can process both small and large graphs efficiently with limited memory. Our partition-based parallel algorithm is shown be dramatically more efficient than the MapReduce algorithm. Finally, we demonstrated the usefulness of rectangle counting through an analysis of real-world networks by rectangle-based clustering coefficient.

References

[1] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, 2008.

[2] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *WEA*, 2005, pp. 606–609.

[3] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[4] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *SODA*, 2002, pp. 623–632.

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *KDD*, 2008, pp. 16–24.

[6] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *PODS*, 2006, pp. 253–262.

[7] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *KDD*, 2011, pp. 672–680.

[8] ——, "Triangle listing in massive networks," *TKDD*, vol. 6, no. 4, p. 17, 2012.

[9] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW*, 2011, pp. 607–614.

[10] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *KDD*, 2009, pp. 837–846.

[11] S. Wasserman and K. Faust, "Social network analysis: Methods and applications," *Cambridge University Press*, 1994.

[12] M. Latapy, C. Magnien, and N. Vecchio, "Basic notions for the analysis of large two-mode networks," *Social Networks*, vol. 30, no. 1, pp. 31–48, 2008.

[13] P. G. Lind, M. C. Gonzalez, and H. J. Herrmann, "Cycles and clustering in bipartite networks," *Physical Review E*, vol. 72, no. 5, 2005.

[14] G. Robins and M. Alexander, "Small worlds among interlocking directors: Network structure and distance in bipartite graphs," *Computational & Mathematical Organization Theory*, vol. 10, no. 1, pp. 69–94, 2004.

[15] P. Zhang, J. Wang, X. Li, M. Li, Z. Di, and Y. Fan, "Clustering coefficient and community structure of bipartite networks," *Physica A: Statistical Mechanics and its Applications*, vol. 387, no. 27, pp. 6869–6875, 2008.

[16] N. Alon, R. Yuster, and U. Zwick, "Finding and counting given length cycles," *Algorithmica*, vol. 17, no. 3, pp. 209–223, 1997.

[17] J. Wang, J. Cheng, and A. W.-C. Fu, "Rectangle counting in large bipartite graphs (long version)," *http://www.cse.cuhk.edu.hk/~jcheng/rect.pdf*, 2013.

[18] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *ICDE*, 2011, pp. 51–62.

[19] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks by h*-graph," in *SIGMOD Conference*, 2010, pp. 447–458.

[20] ——, "Finding maximal cliques in massive networks," *ACM Transactions on Database Systems*.

[21] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *KDD*, 2012, pp. 1240–1248.

[22] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.

[23] J. Wang, J. Cheng, and A. W.-C. Fu, "Redundancy-aware maximal cliques," in *KDD*, 2013, pp. 122–130.

[24] T. R. Halford and K. M. Chugg, "An algorithm for counting short cycles in bipartite graphs," *IEEE Transactions on Information Theory*, vol. 52, no. 1, pp. 287–292, 2006.