Master Thesis

Communication Efficient Triangle Counting

Tim Niklas Uhl

Date of submission: 21 June 2021

First Reviewer:	Prof. Dr. Peter Sanders
Second Reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	M. Sc. Sebastian Lamm

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Counting triangles plays a crucial role in graph processing. Important metrics for analyzing the structure of large real-world networks, such as the clustering coefficient, rely on the number of triangles.

As the size of graphs emerging from real-world applications continuously increases, resulting in inputs with billions of edges, processing them in a reasonable time becomes challenging.

One solution for this is the exploitation of parallelism. Sequential triangle counting algorithms may be directly adapted to shared memory machines, but they are limited in terms of available memory and processors. For handling graphs of massive size, we therefore need to consider distributed memory machine models.

Popular algorithms rely on the MapReduce framework, which results in high amounts of intermediate data required for communication, while other approaches replicate the input among processors. Both techniques may hinder scalability. Therefore the importance of message-passing-based implementations with predictable communication behavior arises. Arifuzzaman et al. [6] propose such an algorithm, but their approach still leaves room for improvement.

We introduce CETRIC, which builds upon this algorithm and reduces the amount of data sent between processors by only requiring communication for counting triangles consisting of cut edges.

Our algorithm uses 1D partitioning to distribute the input and identifies local triangles. They are then removed from the graph, which reduces the neighborhood size of vertices and therefore results in smaller messages.

We suggest additional building blocks of a scalable distributed triangle counter: We speed up set intersection using bit-vectors and further reduce the communication volume by avoiding sending redundant neighborhood information. We also reduce latency introduced by the message startup overhead using an adaptive buffering scheme that aggregates multiple messages.

We present an extensive evaluation of our algorithm using up to 4096 PEs and a large variety of real-world and synthetic networks under strong and weak scaling. We show that our algorithm is $30 - 50\%$ faster than the previously proposed algorithm and scales better with increasing number of processors. Compared to the previous algorithm, it reduces the communication volume by 40%. CETRIC works especially well on web graphs and prepartitioned networks with small cut sizes. Our experiments also focus on load balancing and uncover scaling issues of the previous approach.

Additionally, we outline an approximation algorithm based on CETRIC which uses Bloom filters to further reduce communication.

Zusammenfassung

Das Zählen von Dreiecken spielt eine entscheidende Rolle in der Verarbeitung von Graphen. Wichtige Metriken zur Strukturanalyse von großen Realwelt-Netzwerken, wie beispielsweise der Clustering-Koeffizient, basieren auf der Anzahl der Dreiecke.

Die Größe von Graphen, die aus praktischen Anwendungen heraus entstehen, nimmt stetig zu. Eingaben können Milliarden von Kanten enthalten, sodass eine Verarbeitung in einem angemessenen Zeitrahmen eine Herausforderung darstellt.

Eine Lösung hierfür ist der Einsatz von Parallelismus. Sequentielle Dreieckszähl-Algorithmen können zwar direkt auf Shared-Memory-Maschinen angepasst werden, allerdings mangelt es den Maschinen in der Regel an ausreichend Hauptspeicher und sie verfügen nur über eine begrenzte Prozessorzahl. Um nichtsdestotrotz riesige Eingaben verarbeiten zu können, muss das Distributed-Memory-Maschinenmodell in Betracht gezogen werden.

Bekannte Algorithmen basieren meist auf dem MapReduce-Framework, das zu großen Mengen an Zwischendaten führt, die für die Kommunikation benötigt werden. Andere Ansätze replizieren die Eingabe, doch, genau wie die Verwendung von MapReduce, kann dies die Skalierbarkeit einschränken. Abhilfe schafft das Prinzip des Message-Passing, welches vorhersehbare, feingranulare Kommunikation ermöglicht. Arifuzzaman et al. [6] beschreiben einen solchen Algorithmus, jedoch bietet er noch Verbesserungspotential.

Unser Algorithmus, CETRIC, baut auf diesem Ansatz auf und reduziert die Datenmenge, die zwischen Prozessoren gesendet wird. Dies wird dadurch möglich, dass er Kommunikation nur für Dreiecke benötigt, die ausschließlich aus Schnittkanten bestehen.

Durch 1D-Partitionierung wird der Graph verteilt und jeder Prozessor identifiziert lokale Dreiecke. Durch die Entfernung dieser Dreiecke werden die Nachbarschaften der Knoten reduziert, was auch zu kleineren Nachrichten führt.

Zudem stellen wir weitere Komponenten eines skalierbaren verteilten Dreieckszählers vor: Durch die Verwendung von Bit-Vektoren können wir die Schnittmengen von Nachbarschaften schneller bestimmen und durch die Vermeidung des Sendens redundanter Nachbarschaftsinformationen reduzieren wir das Kommunikationsvolumen weiter. Durch adaptive Pufferung und Aggregierung von Nachrichten reduzieren wir zudem die Latenz, die durch Verbindungsaufbau entsteht.

Wir führen eine umfangreiche Evaluation unseres Ansatzes auf bis zu 4096 Prozessoren und einer Vielzahl an Realwelt- und synthetischen Datensätzen durch. Wir zeigen, dass unser Algorithmus durch die Reduktion des Kommunikationsvolumens zwischen $30 - 50\%$ schneller als der bisherige Ansatz ist und zudem mit zunehmender Prozessorzahl besser skaliert. Im Vergleich zu dem vorherigen Algorithmus reduziert CETRIC das Kommunikationsvolumen um 40%. Unser Ansatz funktioniert insbesondere gut auf Web-Graphen und vorpartitionierten Netzwerken mit kleinen Schnitten. Ein weiteres Augenmerk unserer Experimente liegt auf Lastbalancierung und wir identifizieren Skalierungsprobleme des bisherigen Ansatzes.

Darüber hinaus skizzieren wir einen Approximationsalgorithmus, der auf CETRIC basiert und Bloom-Filter verwendet, um das Kommunikationsvolumen weiter zu reduzieren.

Acknowledgments

I would like to thank Prof. Peter Sanders and Sebastian Lamm for introducing me to the field of distributed memory algorithms and supporting me with valuable discussions and advice during the writing of this thesis. I also thank my family and friends for their support.

I gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`).

Contents

1. Introduction

Graphs are a commonly used abstraction for modeling real-world networks. They may be used to represent a large variety of data relations such as friendships in social networks, road networks, hyperlinks in the world wide web, or protein-protein interaction networks [39]. There exists a vast number of algorithms to analyze the structure and properties of these real-world graphs, but over the last decade, problem sizes algorithms have to deal with are continuously growing. For example, the largest social network, Facebook, had over 2.7 billion active users in the fourth quarter of 2020 [51], and the Common Crawl 2012 web hyperlink graph [36] consists of over 100 billion edges and requires 350GB of memory in uncompressed form.

Due to the size of these networks, processing them in a reasonable time requires the use of parallelism. As the number of processors and available memory on shared memory machines is limited, an important challenge is to design graph algorithms for distributed memory supercomputers with thousands of cores. This is outlined by the popular Graph 500 benchmark [54] which ranks supercomputers based on their performance on fundamental graph algorithms such as breadth-first search and shortest path computation. It also acts as an algorithmic challenge for efficient distributed graph algorithms, using synthetic graphs with up to 2^{46} edges and requiring over 1000TB of memory. A flaw of the Graph 500 benchmark is its limited set of algorithmic problems considered. Triangle counting is an essential algorithm on graphs and has recently gained attention as another benchmark for supercomputers through the Graph Challenge on Subgraph Isomorphism [46].

Triangles are the smallest complete subgraph, and several graph metrics such as the local clustering coefficient [59] need to determine the number of triangles in a graph. Therefore the problem has numerous applications in analyzing complex networks such as social graphs [38] but is also applicable in practice. Becchetti et al. [10] show that analyzing the distribution of the local clustering coefficient may be used to detect spam pages on the web, and Eckmann and Moses [22] identify common topics of web pages using triangle counting. Other applications include database query optimizations [9] and link recommendation [56].

Most algorithms proposed for distributed memory triangle counting either choose a high-level approach and build upon existing graph processing frameworks [44] or the MapReduce framework [52, 43] or they rely on fine-grained optimization of the algorithms for GPUs [29]. Especially when using the MapReduce framework, the communication volume of the algorithm is very high due to large amounts of intermediate data. With an increasing number of processors, this may become a bottleneck, as network speed is a magnitude slower than processing speed. Therefore a pure message-passing-based implementation with predictable communication behavior may be preferred.

Arifuzzaman et al. [6] propose such an algorithm, but while it reduces the amount of communication compared to previous approaches, it still leaves room for improvement.

Our Contribution We introduce CETRIC, a scalable communication efficient triangle counting algorithm that builds upon Arifuzzaman et al.'s algorithm. Compared to previous algorithms, it only requires communication for triangles where each endpoint is located on a different processor. This is achieved by counting triangles locally and removing cut edges from the graph, which reduces the overall communication volume and as a result lowers the running time.

We further speed up the algorithm by using bit-vectors for set intersections and introduce a buffering scheme for messages, which reduces message startup overhead by using message aggregation.

We compare our algorithm to the algorithm by Arifuzzaman et al. [6]. This includes scalability experiments on a large number of real-world and synthetic graph instances with up to 4092 processors. We uncover scalability issues of Arifuzzaman et al. [6]'s algorithm and suggest further improvements for it.

Additionally, we propose an approximation algorithm that further reduces the required communication volume by using Bloom filters.

Structure of this Thesis Our work is structured as follows. In Section 2 we introduce basic definitions and give an overview of the considered machine model. Section 3 gives an overview of the large amount of research in the field of triangle counting and introduces the core building blocks of our algorithm.

In Section 4 we describe CETRIC in detail and how it compares to other distributed triangle counting algorithms. In Section 5 we evaluate different parameters of our algorithm and extensively compare its performance and scalability to PATRIC by Arifuzzaman et al. [6] which is the most notable distributed memory algorithm to date and the base of our algorithm.

Section 6 shows how the communication volume of our algorithm may be reduced even more by using Bloom filters. Section 7 concludes our work and gives directions for future improvements.

2. Preliminaries

In this section we introduce the basic concepts and notations used in this work. We further provide details of the underlying machine model.

2.1. Basic Definitions

Let $G = (V, E)$ be an undirected graph where $V = \{0, \ldots, n-1\}$ and $E \subseteq \binom{V}{2}$ are the sets of vertices and edges, respectively. G has $n = |V|$ vertices and $m = |E|$ edges. For a given vertex $v \in V$ let $N_v(G) = \{u \in V \mid \{u, v\} \in E\}$ denote the *neighborhood* of v.

The *degree* of v is $d_v := |N_v(G)|$, d_{max} is the maximum degree. If the considered graph G is clear from the context, we simply write N_v instead of $N_v(G)$.

Two vertices $u, v \in V$ are called *adjacent* if the edge $\{u, v\} \in E$ exists. An edge $e \in E$ is *incident* to a vertex $v \in V$ if $v \in e$. The vertices incident to an edge are called *endpoints*.

For $V' \subseteq V$ we define the *induced subgraph* $G(V') = (V', E')$ of G, where $E' := \{u, v \in E \mid u \in V' \wedge v \in V'\}$

For a set of three distinct vertices $u, v, w \in V$ we call the induced subgraph $G(\{u, v, w\})$ a *triangle* if and only if it is complete, i.e. each edge $\{u, v\}, \{v, w\}, \{w, u\}$ exists.

Most triangle counting algorithms depend on orienting undirected graphs, to prevent redundant counting of triangles. This is accomplished by using a total ordering $\prec$ on the vertices. Each edge is directed from the lower to higher ranked vertex.

With respect to a total ordering $\prec$ we define the outgoing neighborhood of $v \in V$ as $N_v^+(G) := \{u \in V \mid \{v, u\} \in E \wedge v \prec u\}$ and the incoming neighborhood as $N_v^-(G) := N_v(G) \setminus N_v^+(G)$ and write a directed edge as an (ordered) tuple (u, v) if $u \prec v$. We define the out-degree d_v^+ of a vertex $v \in V$ as $d_v^+ := d_v^+(G) := |N_v^+(G)|$. As before, we do not mention the considered graph explicitly if it is clear from the context.

Based on the definition of Diestel [21] we call a graph *sparse* if the number of edges m is about linear in the number of vertices n and *dense* if m is about quadratic in n.

2.2. Machine Model and Input Format

We consider a system consisting of p processing elements (PEs) numbered $P_0, \ldots, P_{p-1}$, which are connected via a network with full-duplex, single-ported communication. Sending a message of length ℓ from one PE to another takes time $\alpha + \beta\ell$, where α is the time required to initiate a connection and β the subsequent transmission time for sending one machine word. Let the *communication volume* denote the total number of machine words send between processors.

Input Format We assume that input graphs are stored in the *adjacency array* format, which stores the set of neighbors N_v for each vertex v . A graph is represented using an array `head` of size $2m$, which is the concatenation of all neighborhoods for vertices $v \in V$ and an array `first_out` of size $n+1$, where `first_out[v]` stores the first index in `head` containing the neighborhood of v. Then N_v is stored in the index range `first_out[v]`, $\ldots$, `first_out[v + 1] - 1` in the array `head`.

Partitioning the Input We use *1D partitioning*, which means that each P_i is assigned a vertex set V_i, where all sets V_i are disjoint and $V = \bigcup_{i=0}^{p-1} V_i$.

Symbol	Definition
V	vertex set
E	edge set
N_v	$\{u \in V \mid \{v, u\} \in E\}$ for $v \in V$
d_v	$\lvert N_v \rvert$
$\prec$	total ordering on vertex set V
N_v^+	$\{u \in N_v \mid v \prec u\}$ for $v \in V$
N_v^-	$N_v \setminus N_v^+$ for $v \in V$
d_v^+	$\lvert N_v^+ \rvert$
V_i	local vertex set of processor P_i
$\overline{V_i}$	$V_i \cup \bigcup_{v \in V_i} N_v$
∂V_i	$\overline{V_i} \setminus V_i$
$\overline{G_i}$	$\overline{G_i} \coloneqq (\overline{V_i}, E')$, where $E' \coloneqq \{\{u, v\} \in E \mid \lvert\{u, v\} \cap V_i\rvert \geq 1\}$
∂G	$\partial G \coloneqq (V, E')$, where $E' \coloneqq \{u, v \in E \mid u \in V_i, v \in V_j, i \neq j\}$

Table 1: Definitions

P_i stores the neighborhoods for a subsequence V_i of the vertices $\{0, \ldots, n-1\}$. A vertex is called *local* to P_i if it lies in V_i. For a vertex v the rank is defined as $\text{rank}(v) = i :\Leftrightarrow v \in V_i$. In addition to that we assume that the vertices are globally ordered among the processor by vertex ID. This means that if $\text{rank}(v) < \text{rank}(w)$ for vertices $v \in V_i$, $w \in V_j$, $i \neq j$ then w has a higher ID than v.

Vertices u which are contained in a neighborhood N_v for a vertex $v \in V_i$, but are not local to P_i are called *ghost vertices*. Local vertices which are adjacent to at least one ghost vertex are called *interface vertices*.

For processor P_i let $\overline{V_i} \coloneqq V_i \cup \bigcup_{v \in V_i} N_v$ denote the set of all local vertices and ghost vertices and let $\partial V_i \coloneqq \overline{V_i} \setminus V_i$ denote the set of ghost vertices.

We assume that processor P_i only has access to vertices in V_i and their neighborhoods. This effectively means that the local graph of P_i consists of the vertex set $\overline{V_i}$ and all edges $E' \subseteq E$ where at least one endpoint is in V_i (and at most one is a ghost vertex by definition), i.e. $E' \coloneqq \{\{u, v\} \in E \mid \lvert\{u, v\} \cap V_i\rvert \geq 1\}$. We call this graph $\overline{G_i} \coloneqq (\overline{V_i}, E')$.

An edge connecting two vertices $v \in V_i$, $w \in V_j$, $i \neq j$ is called *cut edge*. We say that a distributed memory algorithm is *communication efficient* if the communication volume is proportional to the number of cut edges.

We define the *cut graph* ∂G as the graph only consisting of cut edges, i.e. $\partial G \coloneqq (V, E')$, where $E' \coloneqq \{u, v \in E \mid u \in V_i, v \in V_j, i \neq j\}$.

Table 1 summarizes all notations used in the following.

3. Introduction on Triangle Counting

Counting triangles efficiently in large graphs is an essential building block for analyzing the structure of large networks. Several important metrics for analyzing real-world networks are dependent on the number of triangles in a graph [59, 14, 23]. In the following, we describe three metrics that are directly derived from the number of triangles in G.

The *local clustering coefficient* (LCC) [59] of a vertex $v \in V$ is the density of the subgraph induced by N_v and is defined as follows

$$lcc(v) = \frac{T_v}{\binom{d_v}{2}}$$

where T_v is the number of triangles in G containing v. The intuition behind this is that the LCC describes the probability that two neighbors of a vertex are connected. The average of the local clustering coefficient over all vertices $v \in V$ with $d_v \geq 2$ is called the *clustering coefficient* [59].

The *global clustering coefficient* or *transitivity* [40] measures the global density of triangles with respect to *wedges* in a graph. A *wedge* is a subgraph of three distinct vertices $\{u, v, w\}$ and edges $\{u, v\}, \{v, w\}$ where v is the center vertex. The number of wedges centered at v is Π_v equivalent to the number of neighbor pairs of v, i.e. $\Pi_v = \binom{d_v}{2}$. We therefore define the global clustering coefficient as

$$CC(G) = \frac{3 \cdot \sum_{v \in V} T_v}{\sum_{v \in V} \Pi_v}$$

For example, these metrics allow to analyze properties of time-dependent networks. Akoglu and Dalv [1] show that the local clustering coefficient allows predicting whether ties in phone networks will persist in the future.

To compute these metrics in a reasonable time in massive graphs we need efficient algorithms for counting and listing triangles.

There has been a large amount of research on sequential and parallel triangle counting algorithms. We mainly focus on exact algorithms in sequential and parallel settings in Sections 3.1 and 3.2. In Section 3.3 we also give a brief overview of the most important approximation algorithms.

Other triangle counting algorithms are especially tailored for GPUs [26, 58, 29, 27] and external memory settings. The GPU algorithms mostly focus on exploiting GPUs for fast set intersection.

The external memory algorithm by Chu and Cheng [18] shares minor similarities with our proposed method. Like CETRIC it classifies triangles based on whether they contain ghost vertices with respect to a partition of the input graph and only counts those triangles where all required neighborhood information is available locally. While our algorithm uses this to remove all triangles found locally to reduce neighborhoods and communication, Chu and Cheng write the remaining graph to disk, repartition it and count local triangles iteratively until all triangles have been found.

There also exists a parallel distributed memory algorithm based on matrix multiplication [8], but the authors only perform experiments using up to 256 processors.

All these specialized algorithms are beyond the scope of this work.

We refer the reader to the review by Al Hasan and Dave [2] for a more in-depth look at different triangle counting algorithms, their properties, and applications.

3.1. Sequential Algorithms

A simple brute-force triangle counting algorithm works as follows: for each vertex $v \in V$ we consider all pairs of neighbors $\{u, w\} \in \binom{N_v}{2}$ and check whether u and v are adjacent. This approach has two major shortcomings. On the one hand, we count each triangle three times and therefore do unnecessary work. On the other hand, we need a sophisticated way of storing edges, to allow an efficient adjacency test.

Considering a complete graph K_n with n vertices, it is easy to derive a lower bound on the running time of any triangle counting algorithm. As K_n has exactly $\binom{n}{3}$ triangles, any triangle counting algorithm has an asymptotic lower bound running time in $\Omega(n^3)$ in terms of n or $\Omega(m^{3/2})$ in terms of m [49].

There has been a large amount of work on finding efficient sequential triangle counting algorithms. Schank's Ph.D. thesis [49] and a more recent work by Ortmann et al. [41] give an extensive overview. While the following algorithm EdgeIterator is not the fastest among all, it is the base for many other algorithms in sequential and parallel settings and is easy to implement, which is why it may be considered folklore.

EdgeIterator Like almost all algorithms the algorithm makes the input graph directed in a preprocessing step. EdgeIterator uses a degree-based (total) ordering defined as follows: For $u, v \in V$

$$u \prec v \Leftrightarrow \begin{cases} d_u < d_v & \text{if } d_u \neq d_v \\ u < v & \text{if } d_u = d_v \end{cases}$$

EdgeIterator iterates over all vertices $v \in V$ and their outgoing neighborhood N_v^+. For each directed edge (v, u) the number of vertices adjacent to both endpoints is counted by intersecting N_v^+ and N_u^+. Pseudocode is given in Algorithm 1.

The set intersection in line 6 relies on the neighborhoods being sorted. It is implemented using a procedure similar to the merge phase of merge sort, shown in Algorithm 2.

The running time of EdgeIterator is $\mathcal{O}(d_{max} \cdot m)$ [49].

By orienting the graph, EdgeIterator ensures that each triangle consisting of vertices $\{v, u, w\}$ with $v \prec u \prec w$ is only counted once, while considering vertex v in line 4.

Algorithm 1: EdgeIterator

```
1 for $v \in V$ do
2     sort $N_v^+$ in ascending order
3 $T \leftarrow 0$
4 for $v \in V$ do
5     for $u \in N_v^+$ do
6         $S \leftarrow N_v^+ \cap N_u^+$                // See Algorithm 2 for details
7         $T \leftarrow T + |S|$
```

NodeIterator Another sequential algorithm which may be considered folklore is NodeIterator. Pseudocode of NodeIterator is given in Algorithm 3. As the name suggests, it does not count triangles incident to an edge (u, v), but all triangles containing a vertex $v \in V$. Let $\binom{N_v^+}{2}$ denote the set of all pairs of vertices in N_v^+.

Algorithm 2: Sorted Merge Intersection

```
$(x_0, x_1, \dots, x_{d_v^+-1}) \coloneqq N_v^+$
$(y_0, y_1, \dots, y_{d_v^+-1}) \coloneqq N_u^+$
$S \leftarrow \emptyset$
$i, j \leftarrow 0$
while $i < d_v^+$ and $j < d_u^+$ do
    if $x_i < y_j$ then
        $i \leftarrow i + 1$
    else if $x_i > y_j$ then
        $j \leftarrow j + 1$
    else
        $S \leftarrow S \cup \{x_i\}$
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
```

The algorithm iterates over all vertices in V and for each $v \in V$ considers each pair of outgoing neighbors $\{u, w\} \in \binom{N_v^+}{2}$. For each of these pairs, the algorithm checks if the closing edge $\{u, v\}$ exists in the graph.

To perform this edge query in constant time, the neighborhood of each vertex is usually represented as a hash-map.

Schank [49] states that the running time of NodeIterator is bounded by $\mathcal{O}(d_{\max}^2 \cdot n)$, but an amortized analysis shows that the running time is the same as for EdgeIterator. Based on his experiments Schank concludes, that in practice algorithms based on EdgeIterator outperform NodeIterator, as the use of hashing slows down NodeIterator.

Algorithm 3: NodeIterator

```
$T \leftarrow 0$
for $v \in V$ do
    for $\{u, w\} \in \binom{N_v^+}{2}$ do
        if $\{u, w\} \in E$ then
            $T \leftarrow T + 1$
```

Ortmann et al.'s Algorithm Ortmann et al. [41] describe a common framework to analyze multiple sequential implementation variants. While EdgeIterator uses only constant additional memory for counting triangles, they show that an efficient implementation of an algorithm by Chiba and Nishizeki [17] using $\mathcal{O}(n)$ additional space is faster than all previously proposed algorithms. We call their algorithm K3. The algorithm is similar to EdgeIterator, but instead of intersecting N_v^+ and N_u^+ using sorted intersection, they mark all vertices in N_v^+ using a bit-vector, which allows for intersecting each other neighborhood N_u^+ without having to iterate over N_v^+ multiple times. Pseudocode for K3 is given in Algorithm 4.

The authors show that almost all algorithms investigated, including their proposed algorithm, have a worst-case running time of $\mathcal{O}(a(G) \cdot m)$, where $a(G)$ is the *arboricity* of G, the minimum number of spanning forests needed to cover G.

Algorithm 4: K3 [41]

```
1 $T \leftarrow 0$
2 for $v \in V$ do
3     for $u \in N_v^+$ do  mark[$u$] $\leftarrow$ true
4     for $u \in N_v^+$ do
5         for $w \in N_u^+$ do
6             if mark[$w$] then
7                 $T \leftarrow T + 1$
8     for $u \in N_v^+$ do  mark[$u$] $\leftarrow$ false
```

3.2. Parallel Algorithms

Even carefully tuned sequential algorithms are not enough for today's massive problem instances. Large real-world networks may take up hundreds of gigabytes. For example, the Common Crawl 2012 web hyperlink graph [36] has over 100 billion edges and takes up over 350GB of memory, the Graph 500 benchmark goes even further and uses problem instances with up to 2^{42} vertices and 2^{46} edges, which require over 1125 TB to store them in the adjacency array format. This brings up two major problems: main memory of a single machine may not be sufficient to store the graph and counting may take several hours. For demonstration purposes we ran K3 [41] on a web graph of the UK top-level domain from the LWA dataset [13, 12] consisting of 3 billion edges, which already takes over 50 minutes. A solution for the memory consumption may be to use approximative algorithms, which we discuss in Section 3.3, but in many scenarios determining the exact number of triangles is desired. To deal with such large inputs in a reasonable time, we therefore have to exploit processor and memory parallelism. In this section we consider algorithms using shared memory and distributed memory machine models.

3.2.1. Shared Memory

On a shared memory machine, EdgeIterator may be easily parallelized. As all set intersections in line 6 are independent of each other they may be carried out in parallel and lock-free, as each thread has random access to the whole graph and does not modify it. Shun and Tangwongsan [50] propose a parallel version of EdgeIterator, where the loops in lines 4-5 over vertices $v \in V$ and vertices $u \in N_v^+$ are executed in parallel. To avoid using atomic instructions for incrementing the triangle count T, the number of triangles found by intersecting the endpoints of an edge (u, v) is stored in $T_{(u,v)}$ and the total number of triangles is then aggregated by using a parallel sum operation over the local counts. Orienting the graph with respect to $\prec$ and sorting the neighborhoods may also be parallelized over the vertex set V. They achieve speedups of factor 17-50 for various instances on a 40-core-machine.

Other shared memory implementations [55, 45] share the same main idea with Shun and Tangwongsan's algorithm: They are based on variants of EdgeIterator or NodeIterator and execute the two outer loops of the algorithms in parallel.

3.2.2. Distributed Memory

While triangle counting is easily adaptable to a shared memory setting, as described in the previous section, and achieves a good speedup, its practical use on large instances is limited, as shared memory systems with high number of processors and sufficient amount of memory per processor are not widely available. Real-world networks may be of massive size. As already mentioned, the Graph 500 benchmark [54] aims to simulate large real-world inputs, by generating instances which require up to 1125 TB of memory, while a typical shared memory machine only has 64-128GB of main memory and up to 128 cores.

Handling such large inputs is only possible by exploiting the large amount of available memory and high degree of parallelism of supercomputers. For example, the supercomputer SuperMUC-NG used in our experiments has a total of 304,128 cores and 608TB of main memory. To fully utilize these resources, the development of triangle counting algorithms tailored for a distributed memory machine model is important.

A direct adaptation of the shared memory algorithms discussed above is not possible, because they all require random access to vertex neighborhoods, which is not achievable on a distributed memory machine, where each node only has several gigabytes of memory available.

In the following, we outline the most important distributed memory triangle counting algorithms.

MapReduce-based Algorithms A common framework for distributed parallel algorithms is the *MapReduce* framework [19]. Suri and Vassilvitskii [52] show how to adapt EdgeIterator to the MapReduce framework. They further describe an algorithm that partitions the input graph into overlapping subgraphs which are distributed among reducers. They then use a sequential triangle counting algorithm as a black-box and assign weights to triangles to account for triangles counted multiple times. Park et al. [43] improve this algorithm by reducing redundantly counted triangles and show bounds on the amount of work required per reducer, while increasing the number of MapReduce rounds.

While the algorithms building on the MapReduce framework are easy to implement due to the high level of abstraction, they produce large amounts of intermediate data and in turn require a lot of communication during the shuffle phase, which hinders scalability [48, 43].

Message Passing While MapReduce-based algorithms benefit from abstracting away parallel constructs, they offer no fine-granular control over how data is distributed among machines in a large cluster. Arifuzzaman et al. [5, 6, 7] introduced two message-passing-based triangle counting algorithms, which we describe in the following.

PATRIC PATRIC [5] is based on EdgeIterator (which is oddly called NodeIterator in their publication) and partitions the vertex set of G into p disjoint sets $V_0, \ldots, V_{p-1}$. Processor P_i operates on the induced subgraph $G_i := G(\overline{V_i})$ and counts triangles incident on vertices in V_i using EdgeIterator. By using an induced subgraph, the input graph is therefore partitioned in overlapping subgraphs. The total number of triangles T_i is finally aggregated on a single processor by summing over the local number of triangles T_i for each processor using a collective reduce operation.

This limits the amount of inter-process communication required to a single reduce operation, but the overlapping partitioning scheme hinders scaling of the algorithm with increasing number of processors, as vertices with high degree may be replicated among multiple processors, which increases the amount of memory required.

For PATRIC the authors assume that each PE may initially read arbitrary parts of the graph from disk. Processor P_i needs to know the whole induced subgraph G_i, so it not only requires access to the neighborhoods of each vertex in V_i, but also to the neighborhoods of each ghost vertex. This may not always be easily achievable, as the following example demonstrates. Consider a large social media network with many servers (points-of-presence) all over the world. Each point-of-presence only stores the friendship information of the user in its own geographic region. When we now consider the friendship graph, it is easy to see that it is not feasible to assume that direct access to neighborhoods of ghost vertices is available. Therefore additional communication with other processors is required to gather G_i on a local processor.

PATRIC-NO To address this shortcoming and especially the high memory demand, Arifuzzaman et al. introduce an unnamed variant of PATRIC [6, 7], which partitions the input graph into non-overlapping subgraphs to save memory while requiring more communication. We call this algorithm PATRIC-NO in the following.

PATRIC-NO is based on EdgeIterator. For pseudocode see Algorithm 5. The vertex set of G is partitioned into disjoint subsets $V_0, \dots V_{p-1}$ as before. Each processor P_i is responsible for counting triangles incident on vertices in V_i. For each vertex $v \in V_i$ the algorithms considers all outgoing neighbors $u \in N_v^+$. If u is local to P_i all triangles including edge (u, v) may be found by intersecting $N_v^+ \cap N_u^+$. If u is not local to P_i but to processor P_j, P_i sends N_v^+ to P_j.

This may lead to the neighborhood of vertex v being sent multiple times to P_j, if v is adjacent to multiple vertices in V_j. The algorithm addresses this by keeping track of whether N_v^+ for a vertex $v \in V_i$ has already been sent to P_j and prevents retransmissions. A naive implementation would require $\mathcal{O}(pn)$ additional space on each processor. The authors describe a space-efficient approach which we discuss in detail in Section 4.2.

If P_j receives a neighborhood N_v^+, it extracts all its local vertices $u \in V_j \cap N_v^+$. The neighborhoods of these vertices have to be intersected with the received neighborhood.

The total number of triangles is finally aggregated using a collective reduce and sum operation.

Load Balancing The performance of both algorithms presented heavily relies on the quality of the partitions. If work is distributed unevenly among processors, the running time of a single processor may dominate all others. Arifuzzaman et al. use the same load balancing scheme for PATRIC and PATRIC-NO, which uses a cost function $c(v)$ to estimate the cost for counting triangles incident to vertex $v \in V$. The load balancing procedure is based on a prefix sum. Using the cost function $c(v)$ the cost for each vertex v is estimated and the inclusive prefix sum $C(v) := \sum_{i=0}^{v} c(i)$ is computed. Let $\alpha := \frac{\sum_{v \in V} c(v)}{p}$. A vertex $b_j \in V$ is a *boundary vertex* of P_j if and only if there exists $j \in 0, \dots, p-1$ such that $C(b_j - 1) < j\alpha \leq C(b_j)$. Then processor P_j is assigned the consecutive vertex range $\{b_j, b_j + 1, \dots b_{j+1} - 1\}$. The new local vertices and their corresponding neighborhoods are then reloaded from disk. The authors experimentally evaluated several cost functions and choose the following for PATRIC based on their results:

$$c(v) = \sum_{u \in N_v^+} d_v^+ + d_u^+$$

and

$$c(v) = \sum_{u \in N_v^-} d_v^+ + d_u^+$$

for PATRIC-NO.

The cost function for PATRIC is only an estimation, but PATRIC-NO's cost function exactly measures the number of edge traversals required for counting triangles incident on v [6].

Arifuzzaman et al. describe how to perform this load balancing in parallel. Initially, all local partitions consist of an equal number of vertices. Each PE computes $c(v)$ for all local vertices,

Algorithm 5: PATRIC-NO [6]

```
$T_i \leftarrow 0$
foreach $v \in V_i$ do
    foreach $u \in N_v^+$ do
        if $u \in V_i$ then
            $S \leftarrow N_v^+ \cap N_u^+$
            $T_i \leftarrow T_i + |S|$
        else
            $j \leftarrow rank(u)$
            if $N_v^+$ not sent to $P_j$ yet then
                send $(v, N_v^+)$ to $P_j$
    if receive $(w, N_w^+)$ then
        for $u \in N_w^+$ such that $u \in V_i$ do
            $S \leftarrow N_u^+ \cap N_w^+$
            $T_i \leftarrow T_i + |S|$
while receive $(w, N_w^+)$ do
    for $u \in N_w^+$ such that $u \in V_i$ do
        $S \leftarrow N_u^+ \cap N_w^+$
        $T_i \leftarrow T_i + |S|$
return Reduce($T_i$, SUM)
```

and using a parallel prefix sum operation $C(v)$ is computed for each local vertex. Processor P_i then locates all boundary vertices in its local vertex set V_i and sends them to their respective processor and its predecessor. In the original publication [5], the boundary vertices are located using a linear scan, but in [7] the authors refer to a recursive scan described in detail in [3].

The authors do not account for the cost of gathering the neighborhoods of the new local vertices of P_i, but assume that this information is reloaded from disk. This may not be practical, as disk access may be by a magnitude slower than interprocess communication. For example, each core of the predecessor of the supercomputer used for our experiments, SuperMUC Phase 2, has a memory bandwidth of 137 GByte/s, while the bisection bandwidth of interconnection is 5.1 TByte/s [34].

Note that additional communication is required to evaluate $c(v)$, as a vertex has to know the degree and out-degree of each neighboring vertex, which may be located on another PE.

3.3. Approximative Algorithms

For many applications it suffices to only approximate the number of triangles instead of determining the exact result. Approximation algorithms may reduce both the time and memory requirements of triangle counting when an exact result is not required. The algorithms DOULION and Colorful Triangle Counting both rely on sampling a subgraph of the input and running an exact algorithm on the reduced input instance.

DOULION Tsourakakis et al. [57] introduce Doulion, an edge-sampling-based approximation algorithm that reduces the input size. This allows for trading off running time against exactness and furthermore enables to approximate the number of triangles in a graph which does not completely fit in main memory.

Each edge is sampled with probability p and an exact triangle counting algorithm is used on the reduced input as a black box, giving triangle count T'. The approximated number of triangles is then $T = T' \cdot \frac{1}{p^3}$, as each triangle is left intact with probability p^3. Reducing the number of edges to 10% yields speedups from factor 30 to 130 in their experiments.

Colorful Triangle Counting Pagh and Tsourakakis [42] introduce another approximation algorithm based on sampling. Each vertex $v \in V$ is randomly assigned a color $c(v)$ in the range $1, \dots N$, where $N := \frac{1}{p}$. The reduced input graph then only consists of monochromatic edges, i.e. edges $\{u, v\} \in E$ where $c(u) = c(v)$. This samples each triangle with probability p^2. Again counting the exact number of triangles in the reduced graph T', the approximation is $T = T' \cdot \frac{1}{p^2}$. The authors further show that under assumptions, the estimate produced by their algorithm is strongly concentrated around the exact number of triangles.

Both publications also provide parallel implementations of their algorithms using the MapReduce-framework.

Streaming Algorithms Streaming algorithms are useful for inputs that do not completely fit into main memory. The edges of the graph are processed in a stream of arbitrary order and an approximation of the number of triangles is computed.

Bar-Yossef et al. [9] introduced the first streaming algorithm for triangle counting, which is mostly of theoretical interest. Jha et al. [31] provide a streaming algorithm with space guarantees which only requires a single pass over the edge stream. The algorithm maintains a uniform sample of length-2-paths and checks on each new edge if any of the paths is closed. This gives an approximation of the global clustering coefficient which may then be used to estimate the number of triangles. The authors claim that the algorithm requires $\mathcal{O}(\sqrt{n})$ space to give accurate estimates.

4. Main Contribution

There has been an extensive amount of research in the field of triangle counting but algorithms using message passing which scale well on large supercomputers with high amounts of processors have not received much attention. The most notable work has been PATRIC-NO by Arifuzzaman et al. [6, 7] which we previously discussed in Section 3.2.2. While they analyze the communication cost of their approach and show that they can reduce the number of interprocessor messages compared to a naive algorithm by avoiding to send the neighborhood of a vertex to the same processor multiple times, their algorithm still leaves space for improvement.

We introduce an algorithm called CETRIC (Communication Efficient Triangle Counting) and show that by distinguishing between different types of triangles and using this information to apply a reduction step to the graph, we can reduce the overall communication volume. Using message buffering, we can lower the number of messages by aggregation and thereby reduce the message startup overhead.

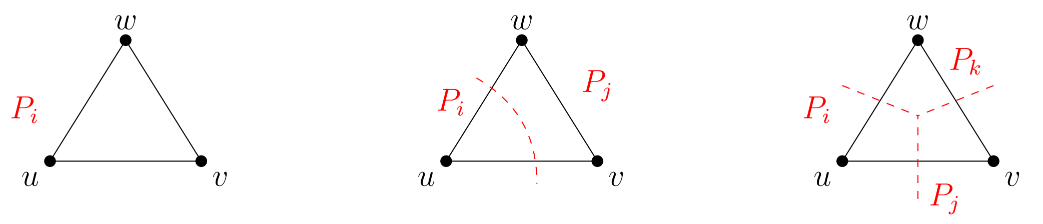
The rest of this section is structured as follows: We first give an overview of the main idea of our exact triangle counting algorithm, going into detail in Section 4.2. Section 4.3 analyzes the local work and communication volume of CETRIC and quantifies the communication reduction compared to PATRIC-NO.

4.1. Overview

Revisiting PATRIC-NO [6], we see that triangles may be classified into three types, depending on the placement of vertices on a processor. Remember that each processor P_i has access to a distinct subset of vertices V_i and the neighborhoods N_v for each vertex $v \in V_i$. Let P_i, P_j, P_k be processors. We call a triangle induced by the set $\{u, v, w\}$ a Type-1 triangle if all vertices are local to a single processor P_i, i.e. if $u, v, w \in V_i$. If any vertex $u \in V_j$, $j \neq i$ and both other vertices $v, w \in V_i$, we call it a Type-2 triangle. If each vertex in the triangle is local to a distinct processor, we call it a Type-3 triangle. The different triangle types are depicted in Figure 1.

Note that Type-1 triangles may always be found without interprocess communication, as all vertices and corresponding edges of the triangle are “known” to P_i. Considering a Type-2 triangle as in Figure 1b, we see that vertices w, v are ghost vertices of P_i. P_i does not have any information about the presence of the edge $\{u, v\}$ and therefore has to communicate with P_j to find the triangle. In contrast to this, P_j knows about all edges of the triangle and no communication is required. In the case of Type-3 triangles, no processor has knowledge about all edges of the triangle and therefore communication is always required.

PATRIC-NO makes no distinction between both cases for Type-2 triangles, as degree-based orientation of edges decides which processor is responsible for counting the triangle. This leads to our key observation: If we can count all Type-1 and Type-2 triangles without communication and then remove all edges which may not be part of a Type-3 triangle from the graph, we can reduce the size of the neighborhoods sent to other processors, thereby reducing the overall message volume. We show that the reduced graph only consists of cut edges and the algorithm is therefore communication efficient.



(a) Type-1 triangle (b) Type-2 triangle (c) Type-3 triangle

Figure 1: Different ways of how a single triangle may be distributed among the processors. Dashed lines indicate processor boundaries.

4.2. Algorithmic Description

CETRIC works in three phases, the *preprocessing phase*, the *local phase* and the *global phase*.

The preprocessing phase makes the input graph directed and sorts neighborhoods, the local phase is responsible for counting Type-1 and Type-2 triangles and reduction of the graph, while the global phase counts all Type-3 triangles and aggregates the results. Pseudocode for our algorithm is given in Algorithm 7.

Preprocessing Phase During the preprocessing phase each interface vertex sends its degree to all neighboring ghost vertices. With this information available, P_i now determines N_v^+ for each $v \in V_i$ using the same degree-based total ordering $\prec$ on the vertices as discussed for EdgeIterator in Section 3.1. This means, that $u \in N_v^+ \Leftrightarrow u \in N_v \wedge u \succ v$. The neighborhoods N_v^+ are then sorted with respect to vertex ID. This is identical to the preprocessing step of PATRIC-NO [6].

Local Phase The local phase is responsible for counting all Type-1 and Type-2 triangles. As discussed in the introduction of this section, this requires no communication. We now go more into detail on how this is accomplished.

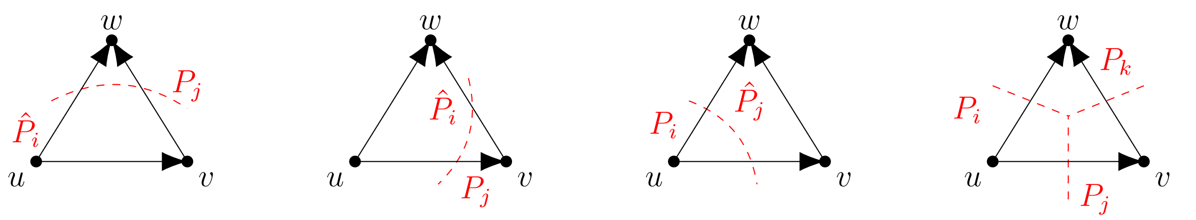
Counting Type-1 triangles is straightforward. A Type-1 triangle only consists of internal edges. The graph $G(V_i)$ induced by all vertices local to P_i only consists of internal edges by definition. If we run EdgeIterator on $G(V_i)$ we therefore count all Type-1 triangles and only Type-1 triangles.

For a Type-2 triangle, we call the processor where two of the triangle's vertices are local to the *designated processor* of the triangle. As discussed before, only the designated processor knows all edges of the triangle. We therefore want to delegate counting a Type-2 triangle to that processor.

Recall that $\overline{G_i}$ is the graph consisting of all edges where at least one endpoint is in V_i and has the vertex set $\overline{V_i}$. The following lemma shows that we can use EdgeIterator as a black-box for the local phase of CETRIC.

Lemma 4.1. *Applying EdgeIterator to each $\overline{G_i}$ counts every Type-1 and Type-2 triangle in G exactly once and counts no Type-3 triangles. A Type-2 triangle is counted by its designated processor.*

Proof. All neighborhoods considered in this proof are defined with respect to $\overline{G_i}$. Let $\{u, v, w\}$ be a triangle and without loss of generality assume that $u \prec v \prec w$. Recall that EdgeIterator iterates over all vertices and over their outgoing neighborhoods, basically iterating over each directed edge and then intersects the neighborhoods of both endpoints.



(a) Case 1 (b) Case 2.a (c) Case 2.b (d) Case 2.c

Figure 2: Case distinctions in the proof of Lemma 4.1. Dashed lines indicate processor boundaries. Only (a)-(c) are counted during the local phase as they are Type-1 and Type-2 triangles. The processor which counts the triangle is marked with a ˆ.

We say that the algorithm *processes* an edge (u, v) if it performs the neighborhood intersection $N_u^+ \cap N_v^+$ and updates the triangle count. We distinguish four cases, which are depicted in Figure 2.

Case 1: $u, v \in V_i$. Then $v, w \in N_u^+$ and $w \in N_v^+$. When examining edge (u, v), EdgeIterator intersects N_v^+ and N_u^+. We then count the triangle, as $w \in N_v^+ \cap N_u^+$. Depending on whether $w \in V_i$ or $w \notin V_i$, this is a Type-1 or Type-2 triangle respectively. If the algorithm processes (u, w) and (v, w) it does not recount the triangle, as $u, v \notin N_w^+$ and $u \notin N_v^+$. Note that all edges of the triangle are processed by P_i.

Case 2: $u \in V_i, v \in V_j, i \neq j$.

Case 2.a: $w \in V_i$. Then $v, w \in N_u^+$, $w \in N_v^+$ with respect to the local graph $\overline{G_i}$. When P_i processes edge (u, v), the same set intersection as in case 1 is performed. The only difference is that N_v^+ only contains vertices in V_i, but as $w \in V_i$, the result is the same and the triangle is found. This is a Type-2 triangle with designated processor P_i. Processing edge (u, w) yields no triangles, as $v \notin N_w^+$. Edge (v, w) is processed by P_i and P_j, but as $u \notin N_v^+$ and $u \notin N_w^+$, the same triangle is not counted twice.

P_j also processes (u, v) but $w \notin N_u^+$ with respect to the local graph $\overline{G_j}$ because u is a ghost vertex and w is not local to P_j. Therefore the triangle is not recounted by P_j but only by the designated processor P_i.

Case 2.b: $w \in V_j$. The Type-2 triangle is only found by processing edge (u, v) as in the previous case, but now only by the designated processor P_j instead of P_i. This can be shown with an argument symmetric to the one given above.

Case 2.c: $w \in V_k, i \neq k \neq j$. This is a Type-3 triangle. Let (x, y) be any edge of the triangle. Without loss of generality assume that x is local to P_i and y is local to P_j. With respect to P_i, N_y^+ then only contains vertices in V_i, as y is a ghost vertex (by the definition of $\overline{G_i}$). Therefore $N_x^+ \cap N_y^+ \subseteq V_i$. As $w \notin V_i$, we do not find the triangle. With the same argument, the triangle is not found by P_j and as we chose (x, y) arbitrarily, the triangle is not found by any processor.

□

After all Type-1 and Type-2 triangles have been counted, all non-cut edges are removed from G, i.e. all edges where both endpoints are contained within the same local vertex set V_i. The remaining graph is isomorphic to the cut graph ∂G.

Global Phase What remains to be counted are the Type-3 triangles from the initial input. As seen before this requires communication. Our algorithm effectively uses PATRIC-NO as a subroutine on the remaining cut graph ∂G resulting after removal of all non-cut edges from the input G.

For this paragraph, all neighborhoods are defined with respect to this reduced graph if not stated explicitly.

Each processor P_i iterates over its local vertex set V_i and for each $u \in V_i$ examines the outgoing neighborhood N_u^+. For each edge (u, v), the endpoint v is located on a different processor P_j. P_i sends the set $S := N_u^+$ to P_j and P_j intersects S and N_v^+. However, if the sending is done naively, this results in a large communication overhead. To be more specific, consider the graph given in Figure 4. Processor P_i would resend N_u^+ to P_j for each edge (u, v_i), $i = 1, \dots \ell$, which results in a high communication overhead due to redundant messages.

Surrogate Approach Arifuzzaman et al. [6] solve this using the so called *surrogate approach*. P_i sends the neighborhood N_u^+ of a local vertex u at most once to each other processor P_j. A receiving processor P_j then completes all computation which requires N_u^+. Therefore P_j has to find all vertices $v \in V_j$ where $v \in N_u^+$ as these are the vertices which would have received the message without the surrogate approach. For a received neighborhood N_u^+ this is the vertex set $N_u^+ \cap V_j$, which is $\{v_1, \dots, v_\ell\}$ in our example. For each vertex $v \in N_u^+ \cap V_j$ the set intersection $N_u^+ \cap N_v^+$ is then performed by P_j.

To ensure that each neighborhood is sent at most once to each processor only $\mathcal{O}(|V_i|)$ memory is required on each P_i. Since each V_i is a consecutive set of vertices by construction and all neighborhoods are sorted, the vertices N_u^+ form consecutive groups of vertices, where each group contains vertices local to P_j where j is larger than the processor index of the previous group. This principle is illustrated in Figure 3. For each $u \in V_i$ the algorithm keeps track of the last processor `last_proc` the neighborhood N_u^+ has been sent to. If P_i processes an edge (u, v) to P_j, it checks if `last_proc` $\neq P_j$. If this is true, P_i sends a message to P_j and sets `last_proc` $\leftarrow P_j$, if not, the message is not sent. Consider that edge (u, v_2) is processed in Figure 3. The rank of v_2 matches `last_proc` and therefore no communication occurs. On examining edge u, v_a this is not true, therefore P_i sends N_u^+ to P_k and updates `last_proc` to P_j.

Therefore keeping track of `last_proc` suffices to ensure that each neighborhood is sent at most once to each processor.

The following lemma shows that due to the reduction of the input to the cut graph ∂G our algorithm avoids that any Type-1 and Type-2 triangles counted during the local phase are recounted and that ∂G contains all Type-3 triangles of G. Here G denotes the initial input of the algorithm.

Lemma 4.2. *The vertex set $\{u, v, w\} \subseteq V$ induces a triangle in ∂G if and only if it is a Type-3-triangle in G.*

Proof. $\Rightarrow$ Assume that $\{u, v, w\}$ induces a triangle in G that is not a Type-3 triangle. Then there exist at least two endpoints of the triangle which belong to the same processor. Without loss of generality assume that these vertices are u and v. Therefore the edge $\{u, v\}$ connects two local vertices and is not contained in ∂G. This implies that at least one edge of the triangle from G is missing in ∂G and the triangle will not be counted.

$\Leftarrow$ Let $\{u, v, w\}$ be a Type-3 triangle in G. Then each edge of the triangle connects vertices located on different processors. Therefore each edge of the triangle is a cut edge and also contained in ∂G.

□

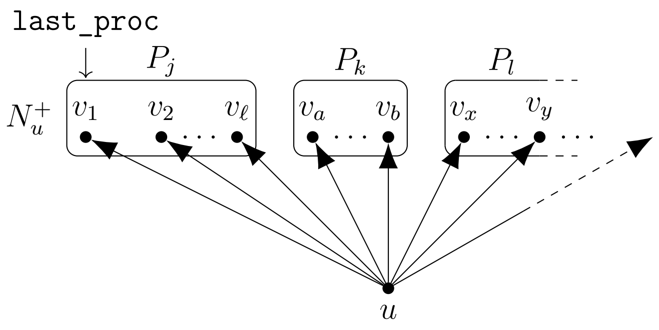


Figure 3: Visualization of the surrogate approach. Each neighborhood N_u^+ is sorted by vertex ID. Vertex and processor IDs increase from left to right. Because each V_i is a consecutive set of vertices, the neighborhood may be divided into consecutive groups, where each group only contains vertices local to a processor P_i and the next group is located on a processor with higher index. N_u^+ is only sent to a processor P_i on examining the first vertex of a group. In this example this only occurs while examining edges (u, v_1) (u, v_a) and (u, v_x).

Together with the correctness of PATRIC-NO [6] this shows that the global phase of CETRIC counts all Type-3 triangles.

By combining the number of Type-1, Type-2 and Type-3 triangles and using a reduce operation over all processors, we obtain the total number of triangles in G.

Combining our triangle classification, Lemma 4.1, Lemma 4.2 and the correctness of PATRIC-NO we get the following corollary.

Corollary 1. *For a given graph G CETRIC counts each triangle in G once and exactly once.*

4.2.1. Further Reducing Message Volume

We can further reduce the size of neighborhoods being sent for counting Type-3 triangles. Assume that the algorithm has finished the local phase, i.e. Type-1 and Type-2 triangles have been counted and all internal edges have been removed. Consider the graph given in Figure 4, where $u \in V_i, v_1, \ldots, v_\ell \in V_j$ and $w \in V_k$. u is adjacent to w, u is adjacent to each $v_i, i = 1, \ldots, \ell$ and each v_i is adjacent to w. The global phase of CETRIC then processes edge (u, v_1).

If we use the algorithm from [6] without adaptation, P_i sends the set $S := \{w, v_1, \ldots, v_\ell\}$ to P_j. Note that sending $v_1, \ldots, v_\ell$ is redundant, as P_j knows all vertices in $N_u^+(G) \cap V_j$, as u is a ghost vertex of P_j and its neighborhood is contained in the local graph $\overline{G_j}$. Therefore we omit all local vertices V_j from a neighborhood being sent to P_j, i.e. P_i sends $S := N_u^+(\partial G) \setminus V_j$ to P_j.

As discussed before, the vertices omitted are required by the surrogate approach, but P_j can reconstruct them from the local neighborhood of ghost vertex u.

4.2.2. Reducing the Number of Messages

Arifuzzaman et al. give no details whether PATRIC-NO uses synchronous or asynchronous sending operations. We use asynchronous sending to prevent blocking. Instead of sending each neighborhood individually, we reduce the number of messages being sent by collecting all messages from P_i designated for a processor P_j in a single send buffer B_j and transmit all send buffers in a batched manner.

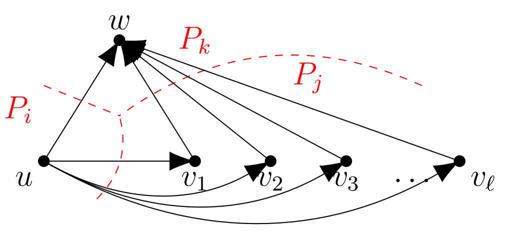


Figure 4: This example graph contains ℓ Type-3-triangles, which are all found by processor P_j. The surrogate approach from [6] prevents that N_u^+ is sent to P_j for each edge (u, v_i). The neighborhood is only sent when processing (u, v_1). Upon receiving the neighborhood, P_j extracts each v_i from the neighborhood and intersects N_u^+ and $N_{v_i}^+$. To reduce the message volume, $v_1, \dots, v_\ell$ may be omitted from the message. The vertices v_i then have to be extracted from the ghost neighborhood $N_v^+(\overline{G_j})$.

Let $v_1, v_2 \in V_i$ be vertices whose neighborhood is sent to P_j. As $N_{v_1}^+ := \{u_1, \dots, u_k\}$ and $N_{v_2}^+ := \{w_1, \dots, w_\ell\}$ have arbitrary size, we separate the neighborhoods using a sentinel element $\perp$. Then send buffer B_j contains the following:

$$B_j = \{v_1, u_1, \dots, u_k, \perp, v_2, w_1, \dots, w_\ell, \perp, \dots\}$$

A receiver P_j may then separate the neighborhood by splitting the set at $\perp$.

Using this approach, the send buffers grow over time, which is not favorable if the memory available per processor is limited. We therefore introduce an additional mechanism. If the size of a message buffer B_j exceeds a certain threshold, we use a single asynchronous send to empty the buffer.

To ensure that new messages may be added to B_j while the send operation has not completed yet we use double buffering: For each target processor P_j we maintain two buffers B_j, B_j'. If a new neighborhood has to be sent to P_j, we add it to B_j. If the size of B_j exceeds the threshold, we swap B_j and B_j' and send B_j' to P_j. New messages for P_j are added to B_j. If B_j overflows, while the sending of B_j' has not completed yet, we block until it has finished. To avoid deadlocks, each processor continuously checks for incoming messages during the global phase.

When each processor has added all its messages to the buffers (and possibly already sent some due to overflowing buffers), we perform a globally synchronized message exchange. Because each processor usually only sends messages to a small subset of communication partners, we use a Sparse-All-To-All operation [28] instead of a full All-To-All. More precisely, each processor sends direct messages to all communication partners asynchronously, and checks for incoming messages until sending has completed. If a processor has completely sent all messages, it hits a non-blocking barrier and continues to check for messages until all other processors have reached the barrier. Pseudocode for this operation is given in Algorithm 6.

As a threshold we use the maximum out-degree $\Delta := \max_{v \in V} |N_v^+(G)|$. Exchanging this threshold between all processors requires an additional AllReduce operation after the preprocessing phase. For readability reasons we omit the double buffering in the pseudocode.

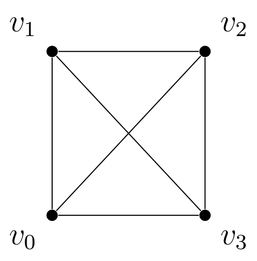
Combining all building blocks discussed in this section, we obtain Algorithm 7. Recall that V_i is the local vertex set of P_i, ∂V_i is the set of all ghost vertices of P_i and $\overline{V_i}$ is the union of both sets.

Algorithm 6: Sparse-All-To-All

```
Input: messages to send $S_0, S_1, \ldots, S_{p-1}$
Output: received messages $R_0, R_1, \ldots, R_{p-1}$
1  foreach $i = 0, \ldots p-1$ do
2      if $S_i \neq \emptyset$ then
3          send_async($S_i$, $P_i$)
4  repeat
5      if message $M$ from $P_j$ available then
6          $R_j \leftarrow M$
7  until all messages sent
8  $b \leftarrow$ non_blocking_barrier()
9  repeat
10     if message $M$ from $P_j$ available then
11         $R_j \leftarrow M$
12 until $b$ reached by all processors
```

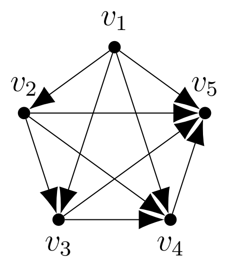
Algorithm 7: CETRIC

```
   /* preprocessing                                                    */
1  for $v \in V_i$ do $N_v^+ \leftarrow \{u \in N_v \mid u \succ v\}$
2  for $v \in \partial V_i$ do $N_v^+ \leftarrow \{u \in V_i \mid v \in N_u \land u \succ v\}$
3  for $v \in \overline{V_i}$ do sort_by_id($N_v^+$)
   /* local phase                                                      */
4  $T_i \leftarrow 0$
5  for $v \in \overline{V_i}$ do
6      for $u \in N_v^+$ do
7          if $u \in V_i$ then
8              $S \leftarrow N_v^+ \cap N_u^+$
9              $T_i \leftarrow T_i + |S|$
10 for $v \in V_i$ do $N_v^+ \leftarrow N_v^+ \setminus V_i$        // remove internal edges
   /* global phase                                                     */
11 for $v \in V_i$ with $N_v^+ \neq \emptyset$ do
12     for $u \in N_v^+$ do
13         $j \leftarrow rank(u)$
14         if $N_v^+$ not sent to $P_j$ yet then
15             $M' \leftarrow \{(v, N_v^+ \setminus V_j\}$
16             $M_j \leftarrow M_j \cup M'$
17 forall the $M_j \neq \emptyset$ do
18     send $M_j$ to $P_j$
19 while receive $(v, N)$ do
20     for $u \in N_v^+$ do
21         $S \leftarrow N_u^+ \cap N$
22         $T_i \leftarrow T_i + |S|$
23 return Reduce($T_i$, SUM)
```



	v_0	v_1	v_2	v_3
$c(v)$	3	3	3	3
$C(v)$	3	6	9	12

(a) Initially $V_0 = \{v_0, v_1\}, V_1 = \{v_2, v_3\}$ PATRIC's load balancing gives $V_0 = \{v_0\}, V_1 = \{v_1, v_2, v_3\}$



	v_0	v_1	v_2	v_3	v_4
$c(v)$	0	7	11	12	10
$C(v)$	0	0	7	18	30

(b) Initially $V_0 = \{v_0, v_1\}, V_1 = \{v_2\}, V_2 = \{v_3\}, V_3 = \{v_4\}$ CETRIC's load balancing gives $V_0 = \{v_0, v_1, v_2\}, V_1 = \{v_3\}, V_2 = \{\}, V_3 = \{v_4\}$

Figure 5: Problems of the load balancing scheme.

4.2.3. Load Balancing

Our load balancing scheme is based on PATRIC-NO. We assign cost $c(v)$ to each vertex $v \in V$ and redistribute the vertices such that each V_i is a consecutive set of vertices, all sets V_i are disjoint and the vertices are globally ordered. The load balancing algorithm tries to redistribute vertices such that cost is equally balanced among processors, i.e. $c(V_i) := \sum_{v \in V_i} c(v) \approx \frac{c(V)}{p}$.

We make minor adaptations to the load balancing procedure from PATRIC-NO as it contains some uncertainties. The first minor problem is that the recursive scan described in [3] is not able to locate boundary vertices which are the first vertex in their partition. This can easily be fixed by adding a sentinel vertex in front of the local vertex set.

In addition, the load balancing scheme does not achieve perfect balance even if it is possible. Consider the example given in Figure 5a. Assume that the graph is a complete graph with four vertices and we use the vertex degree as a cost function. The graph is distributed among two processors such that $V_0 = \{v_0, v_1\}$, $V_1 = \{v_2, v_3\}$. This would also be an optimal assignment with respect to the cost function.

Then $\alpha = \frac{\sum_{v \in V} c(v)}{p} = 6$, so v_1 is a boundary vertex and the load balancing from PATRIC-NO would assign work of cost $c(V_0) = 3$ and $c(V_1) = 9$. We see that work of *at most* size α is assigned to a single processor, which overloads the last processor P_{p-1}.

We solve this problem by adapting the load balancing procedure to assign work of *at least* size α. To achieve this we use an exclusive prefix sum over the vertex cost instead of an inclusive one. We set $C(0) = 0$ and $C(v) = \sum_{i=0}^{v-1} c(i)$ for $v \in V \setminus \{0\}$ and assign a vertex v to processor P_j, where $j = \lfloor \frac{C(v)}{\alpha} \rfloor$. In our example this now gives a balanced partition.

The Load Balancing Scheme Based on our modifications discussed above we obtain the following load balancing procedure.

(i) calculate $c(v)$ for each $v \in V_i$

(ii) calculate the local inclusive prefix sum $C_{\text{local}}(v)$ for each $v \in V_i$ and determine $C(V_i)$.

(iii) calculate the exclusive prefix sum $S(0) = 0$, $S(i) = \sum_{i=0}^{p-1} C_{\text{local}}(V_{i-1})$ for $i > 0$ using a parallel prefix sum operation

(iv) set $C(v_0) = S(i)$ for the first vertex in V_i and $C(v) = S(i) + C_{\text{local}}(v-1)$ for all other local vertices

(v) P_{p-1} determines $\alpha = \lceil \frac{C(n)+c(n)}{p} \rceil$ and broadcasts it

(vi) P_i sends $v \in V_i$ to P_j, where $j = \lfloor \frac{C(v)}{\alpha} \rfloor$

Limits of 1D partitioning A shortcoming of this partitioning scheme is that it may not deal with single vertices where the vertex cost exceeds α. Consider a complete graph with five vertices, given in Figure 5b. Here we use the IDPD cost function (defined below) and four processors. The initial partitioning is $V_0 = \{v_0, v_1\}, V_1 = \{v_2\}, V_2 = \{v_3\}, V_3 = \{v_4\}$ and $\alpha = \frac{40}{4} = 10$. We now employ our modified load balancing scheme. We see that vertices v_0, v_1 and v_2 get assigned to P_0. v_3 gets assigned to P_1, but due to its high cost, the following processor P_2 gets no vertices. Something similar may be observed using PATRIC-NO, where the resulting partition is $V_0 = \{v_0, v_1\}, V_1 = \{v_2\}, V_2 = \{\}, V_3 = \{v_3, v_4\}$. Here the situation gets even worse, because P_3 has to deal with two high cost vertices.

Reassigning other vertices to the processor without work does not give any benefit, as the running time of the algorithm is still dominated by the high cost vertex.

To avoid the imbalance caused by high cost vertices, we considered a tree-based approach, which distributes the neighborhood of high cost vertices among several other processors. More precisely, we rely on partial 2D partitioning, while the main algorithm previously discussed uses 1D partitioning. 1D partitioning only distributes the vertex set among processors and does not subdivide the neighborhood of a single vertex, while 2D partitioning distributes the vertices and neighborhoods among processors. Our main idea is to split the neighborhood of high cost vertices and distribute it among multiple proxy processors which have an empty local vertex set using a binomial broadcast tree. A processor which sends the neighborhood of a local vertex to a high cost vertex then has to split the message and send the parts to the proxy processors of the high cost vertex. Preliminary experiments using a single-level broadcast tree showed that the additionally introduced communication and management overhead did not pay off for a better load balance, which is why we only use the 1D partitioning scheme discussed in detail above for our algorithm.

For our experiments we use the cost functions from PATRIC and PATRIC-NO which we repeat here for clarity.

$$c(v) = \sum_{u \in N_v^+} d_v^+ + d_u^+ \tag{DPD}$$

and

$$c(v) = \sum_{u \in N_v^-} d_v^+ + d_u^+ \tag{IDPD}$$

Arifuzzaman et al. evaluated many other cost functions, but show that for PATRIC the cost function DPD achieves the best load balance on networks with skewed degree distributions [5]. For PATRIC-NO they introduce IDPD. In combination with PATRIC-NO this cost function outperforms DPD [6]. They further show that IDPD exactly measures the cost for local set intersection, but they do not account for communication cost.

This it not easily achievable, as for estimating communication cost we need to identify whether a neighbor of a local vertex is a ghost vertex or not, which is in turn dependent on the resulting partition.

We tried an iterative load balancing scheme, which executed a second load balancing phase after the edge removal step of CETRIC's local phase, but preliminary experiments showed that this does not improve the load balancing and introduces a high running time overhead.

For CETRIC IDPD does not give an exact measure of the required local work. We still consider it a good heuristic and use it together with DPD in our experiments. Preliminary experiments using other cost functions from [5] showed that both cost functions do not only work best for PATRIC and PATRIC-NO, but also for our algorithm.

Our implementation is able to redistribute the graph by sending messages, instead of relying on the availability of the whole graph in the file system of each core of the machine. Relying on the filesystem is not practical, as memory bandwidth may be by a magnitude lower than communication bandwidth, as already discussed in Section 3.2.2.

4.2.4. Implementation Details

In this paragraph we give an insight on some selected details of our implementation, which we omitted in the previous description of CETRIC for brevity, but still have an impact on the performance of the algorithm.

Graph Datastructures Each processor P_i stores its local graph $\overline{G_i}$ using the adjacency array format discussed in Section 2.1 using unsigned 64-Bit integers to represent vertex and edge IDs. Each processor stores the index of the first local vertex and all vertex IDs are translated to local vertex IDs in the range $0, \ldots, \overline{V_i} - 1$ to allow for accessing neighborhoods using a single level of indirection. For local vertices, the global vertex ID may be obtained by adding the index of the first local vertex, the global vertex IDs for ghost vertices are stored explicitly together with the processor the ghost belongs to. Each processor additionally maps the global ID of each ghost vertex to the local ID using a hashmap.

Orienting the input graph with respect to $\prec$ is achieved by scanning over each local neighborhood with a procedure similar to the bucket assignment of inplace quicksort [47], which reorders the array slice representing the neighborhood such that it first contains all incoming and then all outgoing neighbors. For each vertex we store the index in this slice where the first outgoing neighbor is located.

To allow for fast edge removal in the local phase, we additionally store the current degree of each local vertex, while the `first_out` array stores the initial degree. The neighborhood of a vertex v then consists only of vertices `head`[`first_out[v]`]...`head`[`first_out[v] + degree[v]`]. For removing all internal edges of v, we copy all neighbors which are ghosts to the front of the array slice using a single linear scan and update the degree. For a local vertex which is not an interface vertex, i.e. has no ghost neighbors, we can remove all internal edges by setting the degree to zero.

We do not actually free the memory for incoming neighbors and deleted edges, as this would require memory reallocations.

Set Intersections When processing an edge (u, v) we provide two implementation variants for the set intersection. The first variant is identical to the one used in EdgeIterator and PATRIC-NO and uses Sorted Merge Intersection (see Algorithm 2).

The second variant is inspired by K3 by Ortmann et al. [41]. As for each local vertex u, we intersect N_v^+ with N_u^+ for each $u \in N_v^+$, Sorted Merge Intersections scans N_v^+ each time. To prevent this, we use a bit-vector of size $|\overline{V_i}|$ to mark all vertices in N_v^+ and while scanning $w \in N_u^+$ simply check whether w is marked. After processing N_v^+ we reset all flags. For details see Algorithm 4. We call this variant Flag Intersection.

4.3. Theoretical Analysis

In this section, we compare the asymptotic running time and communication volume of our algorithm to PATRIC-NO and show how our algorithm is able to reduce the communication volume.

Let $E_i := \{(v,u) \mid v \in V_i, u \in N_v^+ \cap V_i\}$ be the set of all directed edges on processor P_i, where both endpoints are local to P_i.

In addition, we define the *cut* of a vertex $v \in V_i$ as the set of edges $C_v := \{(v,u) \mid u \in N_v \cap \{V \setminus V_i\}\}$. The set of *outgoing* cut edges is defined as $C_v^+ := \{(v,u) \in C_v \mid v \prec u\}$, i.e. all cut edges directed away from P_i, and the set of *incoming* cut edges is defined analogously as $C_v^- := C_v \setminus C_v^+$.

We further define the outgoing cut with respect to another processor P_j as $C_v^+(j) := \{(v,u) \in C_v^+ \mid u \in V_j\}$. Then $c_u^+ := |C_u^+|$ and $c_u^+(j) := |C_u^+(j)|$.

We write $C_{V_i}^+ := \bigcup_{v \in V_i} C_v^+$ and $C_{V_i}^- := \bigcup_{v \in V_i} C_v^-$ for the overall cut of processor P_i, and define $c_{V_i}^+ = |C_{V_i}^+|$ accordingly. Note, that always $c_u^+ \leq d_u^+$ for $u \in V$.

4.3.1. Local Work

Using the notations introduced above, we first analyze the local work of a single processor P_i using PATRIC-NO and compare it to the local work of CETRIC. For our analysis we assume that both algorithms use Sorted Merge Intersection for intersecting neighborhoods and only focus on the operations required for set intersections.

When either of the algorithms considers an edge $(v,u) \in E_i$, the procedure is identical to EdgeIterator. The neighborhoods N_v^+ and N_u^+ are intersected using Sorted Merge Intersection, which has to traverse both neighborhoods. The running time of set intersection is therefore bounded by $d_v^+ + d_u^+$.

PATRIC-NO For ease of analysis we divide PATRIC-NO into two phases similar to CETRIC. We run the iteration over all edges in lines 2-3 of Algorithm 5 twice. The first iteration is the local phase and we only execute the local triangle counting from lines 4-6, the global phase consists of the second iteration executing the else-branch and performs message receiving and handling.

PATRIC-NO's local phase iterates over all edges in E_i and $C_{V_i}^+$ and processes the edges in E_i. For cut edges in $C_{V_i}^+$ it does not perform any local work, but only sends messages to neighboring PEs. We omit the work required for building messages in the global phase from our analysis.

Therefore the local phase's running time is bounded by $\sum_{(v,u) \in E_i} (d_v^+ + d_u^+) + c_{V_i}^+$.

For each incoming cut edge $(v,u) \in C_{V_i}^-$, the algorithm receives a message containing N_u^+ and intersects it with N_v^+.

This gives the following total required local work for PATRIC-NO

$$\sum_{(v,u) \in E_i} \left(d_v^+ + d_u^+\right) + c_{V_i}^+ + \sum_{(v,u) \in C_{V_i}^-} \left(d_v^+ + d_u^+\right)$$

CETRIC The main idea of CETRIC is to reduce the communication volume by counting Type-1 and Type-2 triangles without communication, which results in smaller messages between communication partners. To achieve this, our algorithm requires more local work than PATRIC-NO.

In addition to intersecting the neighborhoods of the endpoints of each edge in E_i, our algorithm also does the same for each cut edge $(v, u) \in C_v$. Recall that from the view point of P_i, u is a ghost vertex and local to another processor P_j and P_i only has access to edges in $C_u^+(i)$. Therefore processing a cut edge is bounded by $d_v^+ + c_u^+(i)$.

The local work of CETRIC's local phase is then bounded by

$$\sum_{(v,u)\in E_i} \left(d_v^+ + d_u^+\right) + \sum_{(v,u)\in C_{V_i}} \left(d_v^+ + c_u^+(i)\right) .$$

During the global phase, P_i receives messages from other processors P_j. Similar to PATRIC-NO, for each incoming cut edge P_i has to perform a set intersection, but due to the edge removals after the local phase, the size of the sets is reduced. For each incoming cut edge $(v, u) \in C_{V_i}^-$, the partial neighborhood received from u does only consist of cut edges and therefore only has size c_u^+. The neighborhood reduction discussed in Section 4.2.1 omits all cut edges which are directed towards P_i, therefore reducing the size of the received neighborhood to $c_u^+ - c_u^+(i)$. Due to the edge removal, the neighborhood size of v is also reduced to c_v^+.

The local work of the global phase is then given by

$$\sum_{(v,u)\in C_{V_i}^-} \left(c_v^+ + c_u^+ - c_u^+(i)\right) .$$

Combining both phases of CETRIC we get

$$\begin{aligned}
&\sum_{(v,u)\in E_i} \left(d_v^+ + d_u^+\right) + \sum_{(v,u)\in C_{V_i}} \left(d_v^+ + c_u^+(i)\right) + \sum_{(v,u)\in C_{V_i}^-} \left(c_v^+ + c_u^+ - c_u^+(i)\right) \\
= &\sum_{(v,u)\in E_i} \left(d_v^+ + d_u^+\right) + \sum_{(v,u)\in C_{V_i}^+} \left(d_v^+ + c_u^+(i)\right) + \sum_{(v,u)\in C_{V_i}^-} \left(d_v^+ + c_u^+(i)\right) + \sum_{(v,u)\in C_{V_i}^-} \left(c_v^+ + c_u^+ - c_u^+(i)\right) \\
= &\sum_{(v,u)\in E_i} \left(d_v^+ + d_u^+\right) + \sum_{(v,u)\in C_{V_i}^+} \left(d_v^+ + c_u^+(i)\right) + \sum_{(v,u)\in C_{V_i}^-} \left(d_v^+ + c_v^+ + c_u^+\right)
\end{aligned}$$

We see that CETRIC's local phase is more work-intensive, as it has to find all Type-1 and Type-2 triangles by also iterating over the partial neighborhoods of ghost vertices. While PATRIC-NO does no local work on examining a cut edge during the local phase, our algorithm intersects the neighborhoods of both endpoints of each cut edge. This gives an additional work of $\sum_{(v,u)\in C_{V_i}^+} (d_v^+ + c_u^+(i))$.

When the outgoing cut size c_v^+ of a vertex $v \in V$ is small compared to the out-degree d_v^+ our algorithm reduces the amount of work required for processing incoming messages.

To give a crude estimation how the local work of both algorithms compares if the graph has small cuts compared to the degree of cut vertices, we assume that $c_v^+ + c_u^+ \leq \min\{d_u^+, d_v^+\}$ for each cut edges $(v, u) \in C_{V_i}$. Because the global phase works on the cut graph, we save on local work for processing incoming messages. Instead of $\sum_{(v,u)\in C_{V_i}^-} (d_v^+ + d_u^+)$ the last term is $\sum_{(v,u)\in C_{V_i}^-} (d_v^+ + c_v^+ + c_u^+)$ for CETRIC. Under our assumption, this yields a reduction in work required for the global phase.

4.3.2. Communication Volume

We consider the total size of messages sent to communication partners by a single processor P_i. Recall that we call this the *communication volume*, i.e. the number of vertices a processor sends to other PEs. We omit the communication required for load balancing and graph orientation, as they are the same for PATRIC-NO and CETRIC.

PATRIC-NO For each outgoing cut edge $(v, u) \in C^+_{V_i}$, PATRIC-NO has to send the whole neighborhood N^+_v to the processor P_j that u is local to. Recall that $\text{rank}(u)$ defines the processor index the vertex u is local to, i.e. $\text{rank}(u) = j$ if and only if $u \in V_j$. If we send N^+_v to the processor with index $\text{rank}(u)$ for each outgoing cut edge $(v, u) \in C^+_{V_i}$ this would yield a communication volume of $\sum_{(v,u)\in C^+_{V_i}} d^+_v$.

The surrogate approach ensures that N^+_v is only sent once to P_j, which means that the first cut edge to P_j acts as a surrogate for all $c^+_v(j)$ other cut edges to P_j and the communication volume is reduced.

Therefore the communication volume of P_i using PATRIC-NO is given by

$$\sum_{(v,u)\in C^+_{V_i}} \frac{d^+_v}{c^+_v(\text{rank}(u))} \,.$$

CETRIC After the local phase of CETRIC all internal edges are removed from the graph and the resulting graph only consists of cut edges. Therefore for each outgoing cut edge $(v, u) \in C^+_{V_i}$, we only send a small fraction of v's neighborhood to P_j, which has size c^+_v. If neighborhood reduction is used, we can exclude $c^+_v(j)$ additional neighbors of v.

This gives a communication volume of

$$\sum_{(v,u)\in C^+_{V_i}} \frac{c^+_v - c^+_v(\text{rank}(u))}{c^+_v(\text{rank}(u))} \,.$$

As $c^+_v \leq d^+_v$, we see that our algorithm is able to reduce the overall communication volume. The volume is only dependent on the cut size of the graph. If a graph is distributed among processors such that the cut of a vertex is a lot smaller than its outgoing degree, the resulting reduction in communication volume is high. While the local work required is still higher than for PATRIC-NO, we expect our algorithm to outperform PATRIC-NO in practice, as interprocess communication is more time-intensive due to network latency than local computation. We therefore expect the additional local work to pay off.

5. Experimental Evaluation

In this section, we evaluate our algorithm in practice. While we have already shown in theory that CETRIC reduces the communication volume, we investigate the impact on running time on actual data. For that purpose we compare CETRIC to PATRIC-NO and discuss the scaling behavior and impact of load balancing in detail using up to 4096 PEs on the SuperMUC-NG. Note that Arifuzzaman et al. previously only conducted experiments using up to 1000 PEs for PATRIC-NO [5].

In the following, we give details of our methodology. In Section 5.1 we discuss additional improvements CETRIC introduces besides the main communication reduction, while Section 5.2 shows how our algorithm improves upon PATRIC-NO on a large variety of real-world networks and generated instances. Section 5.3 summarizes our experiments.

Experimental Setup We implemented CETRIC in C++. Our implementation is available at `https://github.com/niklas-uhl/parallel-triangle-counting`. We conduct our experiments using two different machines. For a smaller number of processors we use a dual-socket Intel Xeon CPU E5-2683 v4 (2.1 GHz) shared memory machine, with a total of 32 cores and 512GB of DDR4-PC2133 RAM. With hyperthreading enabled we use up to 64 logical cores. The system runs Ubuntu 20.04.2 LTS with kernel version 5.4.0-70-generic. We call this system M1.

For experiments with a large number of processors we use the thin nodes of the SuperMUC-NG supercomputer at the Leibniz Supercomputing Center. The thin nodes consist of 8 islands and a total of 6336 nodes. Each node consists of an Intel Skylake Xeon Platinum 8174 processor with 48 cores. The available memory per core is limited to 2GB. Each node runs the SUSE Linux Enterprise Server (SLES) operating system.

Our code is compiled using optimization level `-O3`. On the shared memory machine we use g++-9.3.0 and OpenMPI 4.0.3, on the SuperMUC-NG we use g++-9.2.0 and Intel MPI 2019 compiled with gcc.

The authors of PATRIC-NO do not provide an implementation of their algorithm, therefore we use our own tuned implementation, which includes our message buffering scheme, adapted load balancing and optionally may use Flag Intersection and neighborhood reduction as described before. As this reduces the differences between PATRIC-NO and CETRIC implementation wise, it allows for a more direct comparison of both competitors.

Datasets We evaluate CETRIC on a large variety of real-world and synthetic instances. All instances are listed in Table 2. We visualize the degree distribution of all instances in Appendix A.

We use live-journal and orkut from the SNAP dataset [35], a complete snapshot of follower relationships of the twitter network [33][1] and wiki-links-en and friendster from the KONECT collection [32][2]. We further include large web graphs from the Laboratory for Web Algorithmics dataset collection (LWA), namely uk-2007-05 and webbase-2001 [13, 12][3].

The uk-2007-05 graph has a size of 50 GB and consists of over 3 billion edges and is the largest real-world graph used in our experiments. It is the result of a crawl of the `.uk` domain. webbase-2001 has been originally obtained from the 2001 crawl performed by the WebBase crawler. While having more vertices than uk-2007-05, it only has around 855 million edges.

[1]available for download at `http://an.kaist.ac.kr/traces/WWW2010.html`
[2]available for download at `http://konect.cc`
[3]available for download at `http://law.di.unimi.it/datasets.php`

instance	n	m	memory	source	graph family
live-journal	4,847,571	42,851,237	691 MB	SNAP [35]	social
orkut	3,072,441	117,185,083	1.8 GB	SNAP [35]	social
twitter	41,652,230	1,202,513,046	19 GB	[33]	social
wiki-links-en	13,593,032	334,591,525	5 GB	KONECT [32]	web
friendster	68,349,466	2,586,147,869	28 GB	KONECT [32]	social
uk-2007-05	105,896,555	3,301,876,564	50 GB	LWA [13, 12]	web
webbase-2001	118,142,155	854,809,761	14 GB	LWA [13, 12]	web
europe	18,029,721	22,217,686	477MB	DIMACS [20]	road
usa	23,947,347	28,854,312	623MB	DIMACS [20]	road

Table 2: Real-world graph instances used in our experiments.

We use road networks of europe and usa made available during the DIMACS challenge on shortest paths [20]. Road networks typically have low average degree and a uniform degree distribution.

Whenever the graphs are directed, we interpret each edge as undirected. We remove vertices with no neighbors from the input.

For generating synthetic instances we use the distributed graph generation framework KaGen[4] and generate 2D random geometric graphs (RGG) and random hyperbolic graphs (RHG). The RGG generator works by placing n vertices at uniform random locations in the unit plane. Two vertices are adjacent if their euclidean distance is less than a given radius r. We set $r = r_{\text{coeff}}\sqrt{\frac{\ln n}{n}}$ with $r_{\text{coeff}} = 0.55$ as this ensures that the graph is almost always connected [4] and is used by Funke et al. [25]. We write $RGG(n, r_{\text{coeff}})$ to identify an RGG graph with given number of vertices and radius.

The RHG model places vertices randomly on a disk with fixed radius, such that many vertices are concentrated around the center. Two vertices are adjacent if their hyperbolic distance is below a certain threshold. The generated graphs follow a power-law degree distribution and may be parameterized using the power-law exponent γ, the desired average degree $\overline{d}$ and the number of vertices n. We use the parameters used in [25] and set $\overline{d} = 16$ and use $\gamma \in \{2.8, 3.0\}$. We write $RHG(n, \overline{d}, \gamma)$ to identify an RGG graph with given number of vertices, average degree and power-law exponent γ.

The graph generators from KaGen work without communication and partition the vertex set of the generated graph and already contain locality. We therefore do not use our load balancing scheme in conjunction with synthetic instances, but rely on the partitioning provided by the generators. This maintains locality of the input and allows us to evaluate our algorithm independent of load balancing.

We further use the recursive matrix graph model (R-MAT) by Chakrabarti et al. [16]. It is well known, as it is used in the popular Graph 500 benchmark [54] for generating large instances. An R-MAT graph with n vertices and m edges is generated by recursively subdividing the adjacency matrix into four equally sized sectors, where the four sectors are assigned probabilities a, b, c, d with $a + b + c + d = 1$. Each edge is sampled independently by descending into any of the four sectors with the assigned probability. This is repeated recursively until the sector has size 1.

We use the generator from [30] and the parameters from the Graph 500 benchmark, i.e. $a = 0.57, b = 0.19, c = 0.19, d = 0.05$ and set $m = 16 \cdot n$.

[4]https://github.com/sebalamm/KaGen [25]

Methodology If not stated otherwise, we measure running time of PATRIC-NO and CETRIC after load balancing has been performed because the load balancing phase of both algorithms is identical. Our measurements include the preprocessing, local and global phase for CETRIC and preprocessing and the single main phase for PATRIC-NO. We call this the *algorithm core*. On each processor we measure the execution time of all phases combined using `MPI_Wtime` and report the maximum over all processors. For fine granular per processor measurements we use `std::chrono::high_resolution_clock` from the C++ standard library.

We measure communication volume by counting the number of vertices a single processor P_i sends to its communication partners and report the total sum over all processors. This includes sentinel elements sent. We only count vertices sent during the main phase of the algorithms, i.e. do not include communication required during the load balancing and preprocessing phase and do not include the cost for the final reduction operation, because the phases are identical for PATRIC-NO and CETRIC. Preliminary experiments have shown that the additional communication volume from these phases is smaller than the communication volume of the global phase on most large instances. Our analysis therefore only focuses on communication during the core algorithm.

To get a better understanding of how much each algorithm phase contributes to the running time, we refer the reader to Appendix B.

We consider two base algorithms in our experiments, PATRIC-NO and CETRIC. We further evaluate our tuned implementation of PATRIC-NO, which includes Flag Intersection and neighborhood reduction discussed in Sections 4.2.4 and 4.2.1. We call this variant PATRIC-NO+. For comparability reasons, all algorithm variants use our message buffering scheme from Section 4.2.2. For all experiments on real-world data, we use the load balancing procedure discussed in Section 4.2.3 and use either DPD or IDPD as cost functions. For graphs generated using distributed graph generators, we use the vertex partition provided by the generator and skip the load balancing step.

5.1. Preliminary Experiments

Besides the main idea of CETRIC, avoiding communication for counting Type-2 triangles, our algorithm introduces three other building blocks of a scalable distributed triangle counter. The first one is the reduction of message sizes by omitting locally available information, the second idea is to use a bit-vector for consecutive set intersections with one fixed set, as this prevents scanning over the same set multiple times. As a third technique, we propose aggregating multiple messages designated for a single processor in message buffers, which reduces the total number of messages and saves on startup overhead. These building blocks may also be directly incorporated in PATRIC. In this section, we evaluate the impact of these techniques on message volume and running time and how they improve CETRIC.

5.1.1. Neighborhood Reduction and Set Intersection

As discussed before, we can further reduce the amount of data being sent between processors during the global phase. Instead of sending $N_u^+(\partial G)$ to P_j, we omit all vertices local to P_j and only send $N_u^+(\partial G) - V_j$.

Additionally, we discussed two variants of how the neighborhood intersection in CETRIC may be performed in Section 4.2.4. The first variant, Sorted Merge Intersection, works similar to the merge phase of merge sort and for each intersection performs a simultaneous linear scan over both sets and is also used in EdgeIterator and PATRIC.

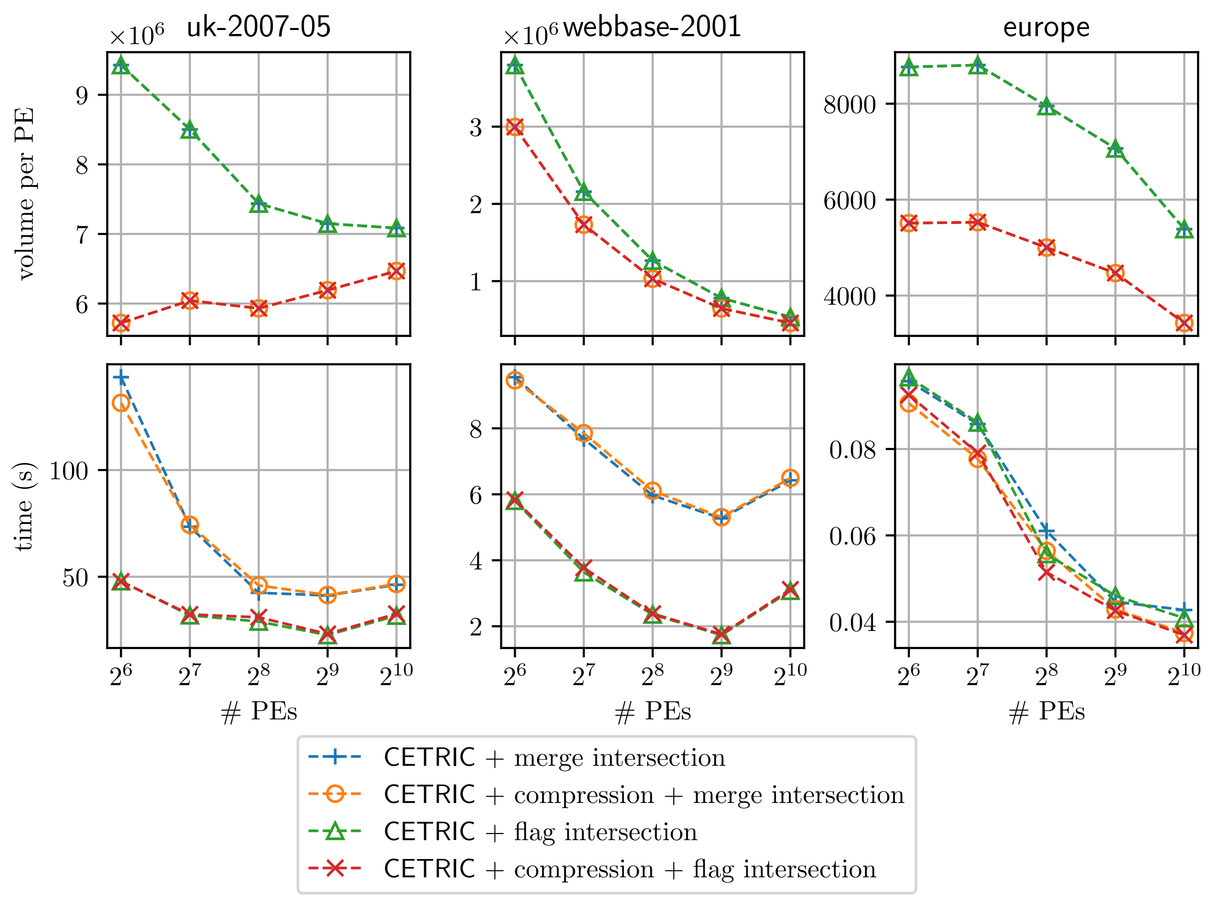


Figure 6: Communication volume and running time of CETRIC-IDPD with neighborhood reduction and different implementation variants of set intersection using 64 to 1024 PEs of the SuperMUC-NG.

We propose a second variant called Flag Intersection inspired by K3, which marks the vertices of one neighborhood using a bit-vector and for the other neighborhood only checks for each element if the flag is set. This reduces the time required for set intersection if the first set is fixed and intersected with multiple other sets, as we only have to scan over the second set.

Using CETRIC with IDPD as cost function for load balancing we report the communication volume per PE and the running time with and without using neighborhood reduction and for the implementation variants of the set intersection. We show the results for the real-world instances uk-2007-05, webbase-2001 and europe in Figure 6, but observed similar results for other instances.

On the considered instances, neighborhood reduction reduces the communication volume by $20 - 36\%$ on average, but only on europe this results in a lower running time.

Flag Intersection does not change the communication volume, but halves the running time on uk-2007-05 and webbase-2001. On europe Flag Intersection only slightly affects the running time. We assume that this is due to the low degree of road networks.

Even though in our experiments neighborhood reduction does not influence the running time, we assume that its effects become visible on distributed memory machines with lower communication bandwidth than the SuperMUC-NG.

Based on these results we use neighborhood reduction and Flag Intersection for the rest of the experiments in this section and all following experiments in combination with CETRIC.

5.1.2. Message Buffering

In Section 4.2.2 we introduced the concept of message buffering. Each processor aggregates all messages designated for processor P_j in a message buffer B_j, instead of immediately sending them. By sending multiple messages in a batch, we only need to account for communication startup overhead once per batch instead of for each single message. A shortcoming of this approach is that additional memory is required to store the message buffers. As discussed before, we address this by emptying buffers if their capacity exceeds a certain threshold.

Message buffering has almost no effect if executed on a shared memory machine, because message exchange happens in memory, and the message startup overhead is low. On a distributed system the startup overhead is higher, because it is dependent on the network bandwidth.

In Figure 7 we show the impact of message buffering on uk-2007-05 using CETRIC and IDPD as cost function. We compare the total number of messages sent for increasing number of processors using no buffering, as suggested for PATRIC-NO in [5], using a threshold of $\Delta = \max_{v \in V} |N_v^+(G)|$, and using a threshold of ∞. Based on our previous experiments we use neighborhood reduction and Flag Intersection.

Our experiments support the assumption that message buffering may significantly reduce the number of messages and also has a direct impact on the total running time. For example, if the threshold is set to Δ using 512 PEs, the number of required messages is reduced by 99% compared to the algorithm without buffering. The resulting reduction in running time is 10%. Especially with increasing number of processors, using buffering with a threshold of Δ clearly outperforms that variant without buffering.

If the threshold is set to ∞, the total number of messages is bounded by $\mathcal{O}(p^2)$ as each processor sends at most one message to each other processor in a globally synchronized communication step (see Algorithm 6 for details). The dotted line in Figure 7 indicates p^2. We see that the number of messages for a threshold of 0 is close to this bound using 64 PEs, but with increasing number of processors does not grow as fast as p^2. This does not only reduce the message startup overhead. If messages may be sent at any time, each processor has to check for messages continuously, which introduces further latency. If communication only happens in a single globally synchronized step, this checking may be omitted. This does not come for free, as the large buffers require lots of memory.

We observed similar results on other graph instances.

As the per core memory on the SuperMUC-NG is limited to 2GB, we use Δ as a buffering threshold for the rest of our experiments. For a fair comparison, we use the same buffering scheme and threshold for all algorithms.

5.2. Scaling Experiments

We analyze the scalability of our algorithm in terms of strong and weak scaling. For *strong scaling* experiments we choose an input instance with fixed and measure the running time over all processors with increasing number of processors. *Weak scaling* measures the variation of running time for a fixed problem size *per PE*, i.e. we scale the problem size proportional to the number of processors. We first focus on strong scaling experiments using real-world and synthetic graph instances in Section 5.2.1 using up to 4096 PEs. For the weak scaling experiments in Section 5.2.2 we use inputs with up to 2^{32} vertices on 4096 PEs.

Based on our preliminary results in Section 5.1 we use CETRIC in conjunction with neighborhood reduction and Flag Intersection. We also use message buffering with the buffering threshold set to $\Delta = \max_{v \in V} |N_v^+(G)|$. To allow for a fair comparison of the algorithms in-

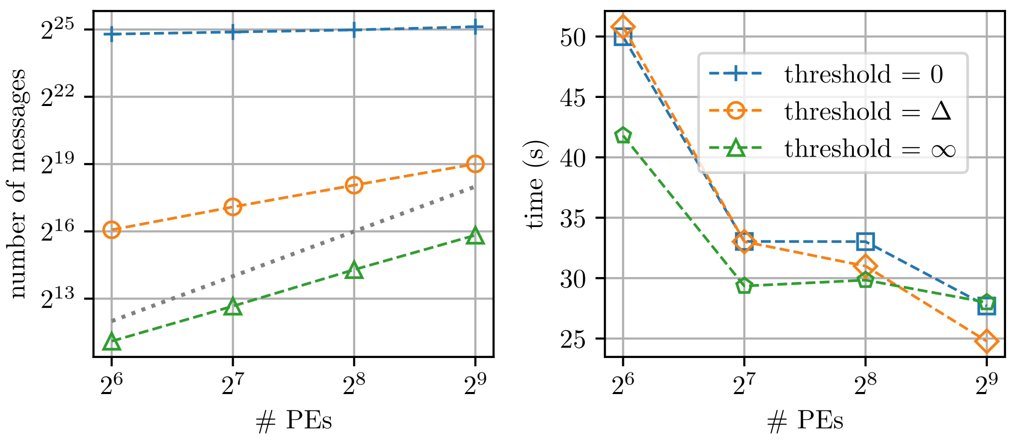


Figure 7: Total number of messages and resulting running time of CETRIC-IDPD on uk-2007-05 for different buffering thresholds on the SuperMUC-NG.

dependently of the buffering scheme, we use the same buffering approach for PATRIC-NO and PATRIC-NO+.

5.2.1. Strong Scaling

We first give a brief overview of our algorithm performance by comparing it to PATRIC-NO on the shared memory machine M1 using up to 64 PEs. After that we perform a more in-depth evaluation on several selected instances using the SuperMUC-NG with up to 4096 PEs. This includes a thorough discussion on load balancing, scalability and on the several improvements techniques introduced by our algorithm.

We evaluate our algorithm by comparing its communication volume and running time to PATRIC-NO. We performed our initial experiments on the shared memory machine using 2 to 64 PEs. The following plots show the average communication per PE on the left, the total running time of the algorithm core (all phases excluding load balancing) is shown on the right. We evaluate all algorithms using IDPD and DPD as cost functions. We use three families of real-world instances: social networks, web graphs and road networks.

Figure 8 shows our results for social networks. At a first glance we notice that the development of communication volume is similar for all instances. We see that on all inputs our algorithm is able to reduce the communication volume. Using 2 PEs, the volume is reduced by up to 95% on twitter, with similar reduction on other instances. With increasing number of processors the amount of reduction decreases, but we still achieve an average reduction of $30 - 37\%$ over all measured values. More importantly, CETRIC is also faster than PATRIC-NO in most cases. Again considering twitter as an example, we achieve a running of time $623s$ using 32 PEs, which is a reduction by 25% compared to PATRIC-NO using the same number of processors.

On live-journal we see that CETRIC outperforms PATRIC-NO up to 8 PEs, but then the competitor catches up. We further see that IDPD does not scale as well as DPD. On orkut our algorithm hits a scaling wall at 64 PEs. We will focus on these two instances more in detail below.

The reduction of the communication volume also has a direct influence on running time, even though on the shared memory machine communication latency is a lot lower, because no networking is required. While Arifuzzaman et al. intensively discuss different cost functions, on lower number of processors and large graphs, the difference is negligible. Especially using up to 8 cores, which is typical for end user systems, CETRIC greatly improves the running time.

On all social instances our algorithm reduces the communication by a high margin for up to 32 PEs, but then the difference between CETRIC and PATRIC-NO becomes small.

Figures 9 and 10 show our results on web graphs and road networks. Here we see, that our algorithm performs even better than on social instances. On uk-2007-05 the communication volume is reduced by 86% on average, the running time is reduced by 60% on average. On uk-2007-05 and webbase-2001 CETRIC clearly outperforms PATRIC-NO. On these instances, we also see that the cost function has almost no influence on communication volume and running time. Only on wiki-links-en when using more than 32 PEs, PATRIC-NO catches up with our algorithm.

On road networks, our algorithm clearly outperforms PATRIC-NO in terms of communication volume and running time. On europe the choice of cost function again has almost no effect, while on usa the cost function IDPD leads to large variances in running time with increasing number of processors.

In the following we provide a more detailed discussion of our algorithm and take a closer look at load balancing and scalability with a higher number of PEs as well as including PATRIC-NO+ in our experiments.

We focus on the large web graphs uk-2007-05 and webbase-2001, the road networks and live-journal and orkut. We evaluate the scaling behavior of these graph families and of additional generated graphs from 64 to up to 4096 cores on the SuperMUC-NG. To give a better understanding of how good CETRIC works, we further compare to PATRIC-NO+ which is our tuned variant of PATRIC. Due to the memory limitations of SuperMUC-NG (2GB per core), we were not able to conduct these experiments on large social networks such as twitter, as they produce large amounts of intermediate data to represent ghost vertices and their neighborhoods, such that the local data structures do not fit into a single core's main memory.

Web Graphs We now give a detailed comparison of CETRIC, PATRIC-NO and PATRIC-NO+ on uk-2007-05 and webbase-2001. Recall that uk-2007-05 is the largest instance used in our experiments. We evaluate our algorithm and its competitors on these two graphs.

Figure 11 reports the results of our strong scaling experiments on both graphs. Considering uk-2007-05 we see that for both cost functions, CETRIC is able to reduce the communication volume compared to PATRIC-NO. For IDPD the communication volume is reduced by 40% on average, for DPD even by 50%. Using 64 PEs and IDPD as cost function, we are able to reduce the volume by 77% and the amount of reduction decreases with increasing number of processors. Still, using 2048 PEs, CETRIC-IDPD requires 10% less communication than PATRIC-NO-IDPD. This may be explained by the fact that the number of cut edges increases when the graph is distributed among a larger number of processors. As CETRIC achieves its communication reduction by omitting non-cut edges from the messages, its impact on the communication volume becomes less if the cut size increases.

We included PATRIC-NO+ in our experiments because it incorporates neighborhood reduction and Flag Intersection into the algorithm by Arifuzzaman et al. [6]. This allows us to investigate the impact of CETRIC's main idea, avoiding communication for Type-2 triangles independent of other improvements included in our algorithm.

While the neighborhood reduction from PATRIC-NO+ slightly lowers the volume, we see that CETRIC outperforms it in terms of communication volume and running time.

We achieve the best running time on uk-2007-05 using 512 PEs and CETRIC-IDPD, where the running time is 25 seconds. Compared to that, PATRIC-NO using the same cost function takes 31 seconds.

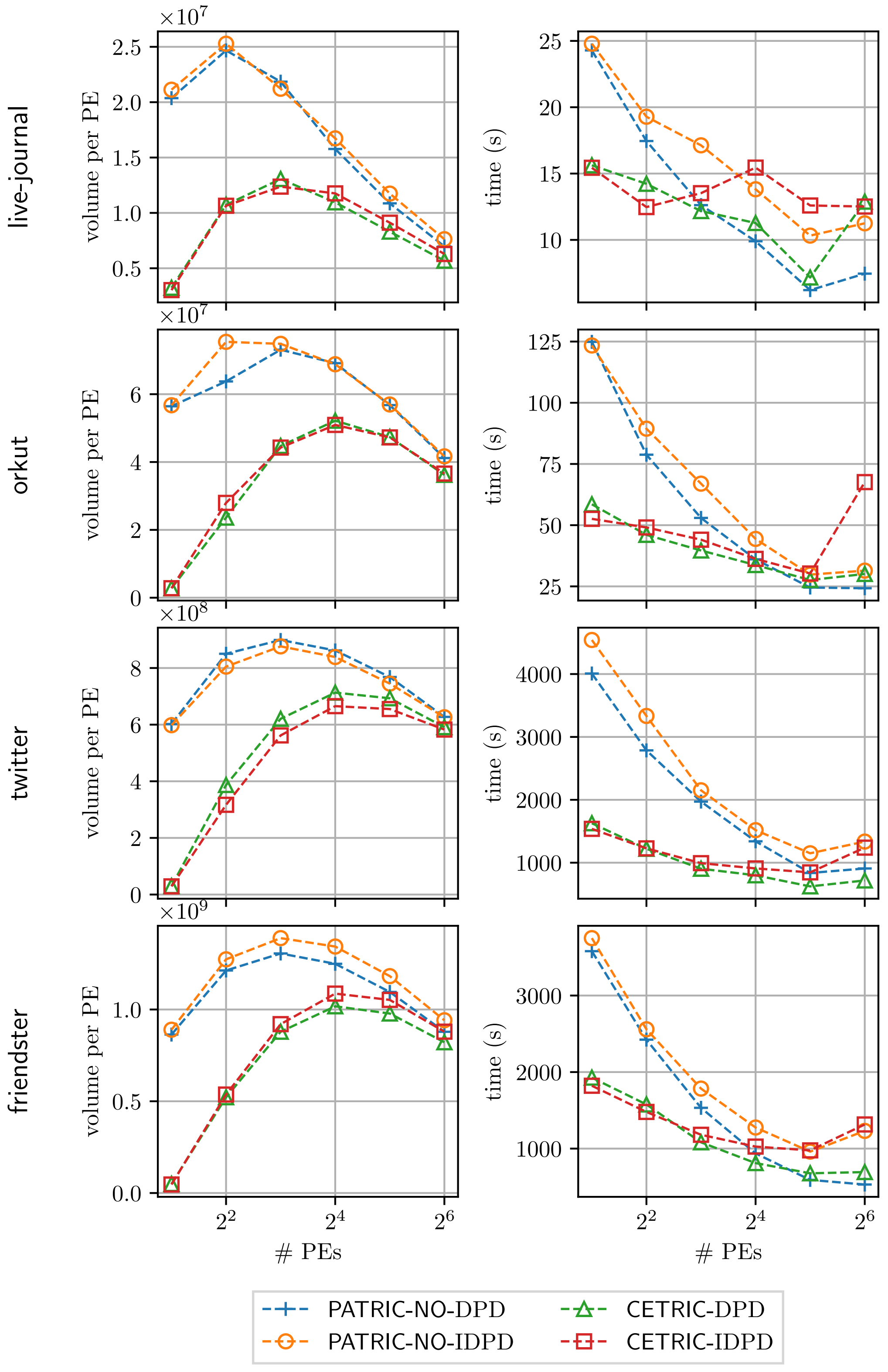


Figure 8: Strong scaling on social networks using 2 to 64 PEs on M1.

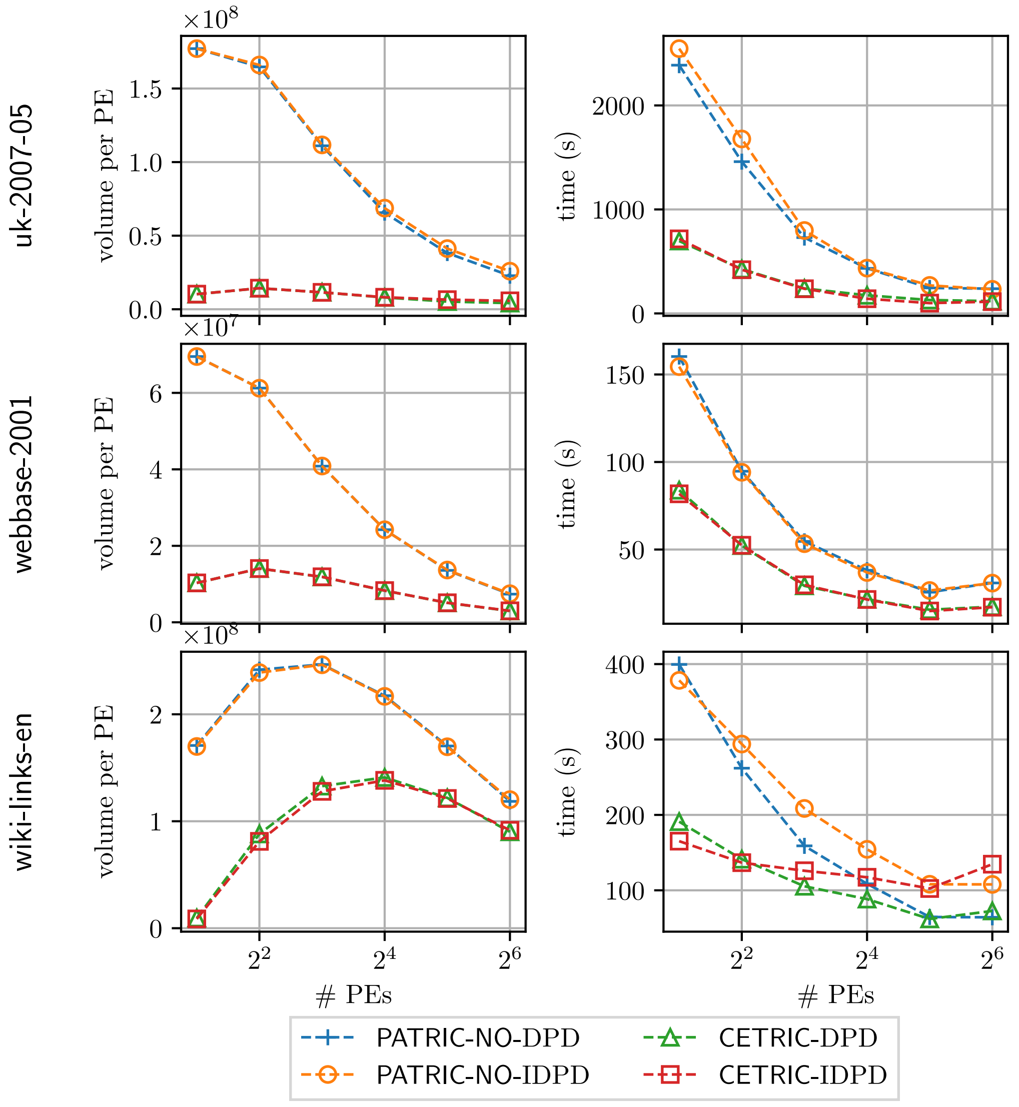


Figure 9: Strong scaling on web graphs using 2 to 64 PEs on M1.

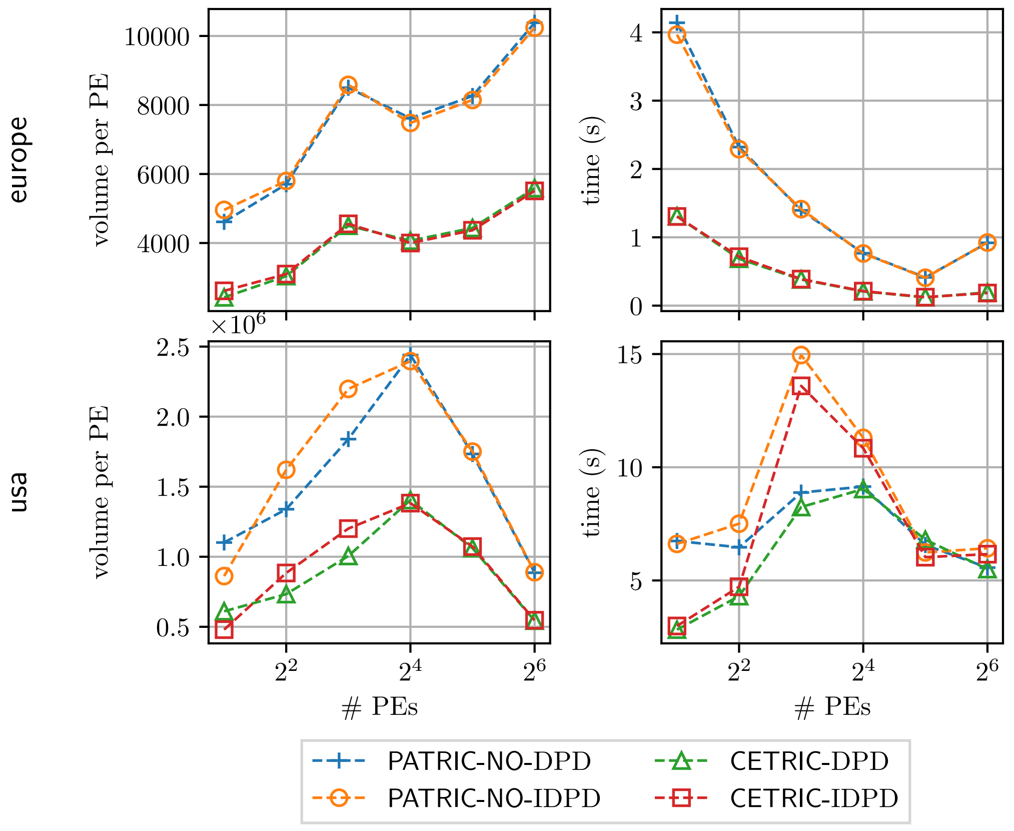


Figure 10: Strong scaling on road networks using 2 to 64 PEs on M1.

Up to a processor count of 512 PEs, CETRIC-IDPD clearly outperforms its competitors. With higher number of processors, all configurations using IDPD hit a scaling wall.

Further investigations have shown that this is due to a load imbalance. Using 4096 PEs there is one PE which dominates the running time, caused by a single vertex located on it.

This vertex has high degree but is also adjacent to more high degree vertices. Recall that the vertex cost is defined as $c(v) = \sum_{u \in N_v^-} d_u^+ + d_v^+$ using IDPD. The considered problematic vertex has both high in-degree and is adjacent to vertices with high out-degree on their own, which results in a high cost. As discussed before in Section 4.2.3, the load balancing procedure treats each vertex as an atomic task and assigns it to a PE P_i. Besides the problematic vertex this PE only gets assigned a small number of low cost vertices. Therefore, almost all neighbors of the problematic vertex are ghost vertices and the processor P_i sends and receives a large number of messages due to its high degree.

The problem is depicted in Figure 12. For each PE P_i we measure the accumulated cost $c(V_i) = \sum_{v \in V_i} c(v)$ using IDPD as cost function $c(v)$. Recall that the goal of the load balancing scheme is to achieve $c(V_i) \approx \alpha := \frac{C(v)}{p}$ local cost. For each PE we report the imbalance, i.e. the relative error $\epsilon = \left|1 - \frac{C(V_i)}{\alpha}\right|$ of the actual accumulated cost of the PE compared to the desired cost α. The left y-axis of each subplot describes this error, the right y-axis shows the message volume sent by each processor. The left plot shows the results for 512 PEs, the right plot for 4096 PEs. We see that in the case of 4096 PEs the processor with index 3162 has a send volume which is by a factor of 775 higher than the average send volume per PE. This imbalance causes scalability issues. We notice that this correlates with the imbalance error. For 512 PEs we see that the imbalance does not occur.

Surprisingly, using DPD for load balancing, the scaling behavior is a lot better. As DPD

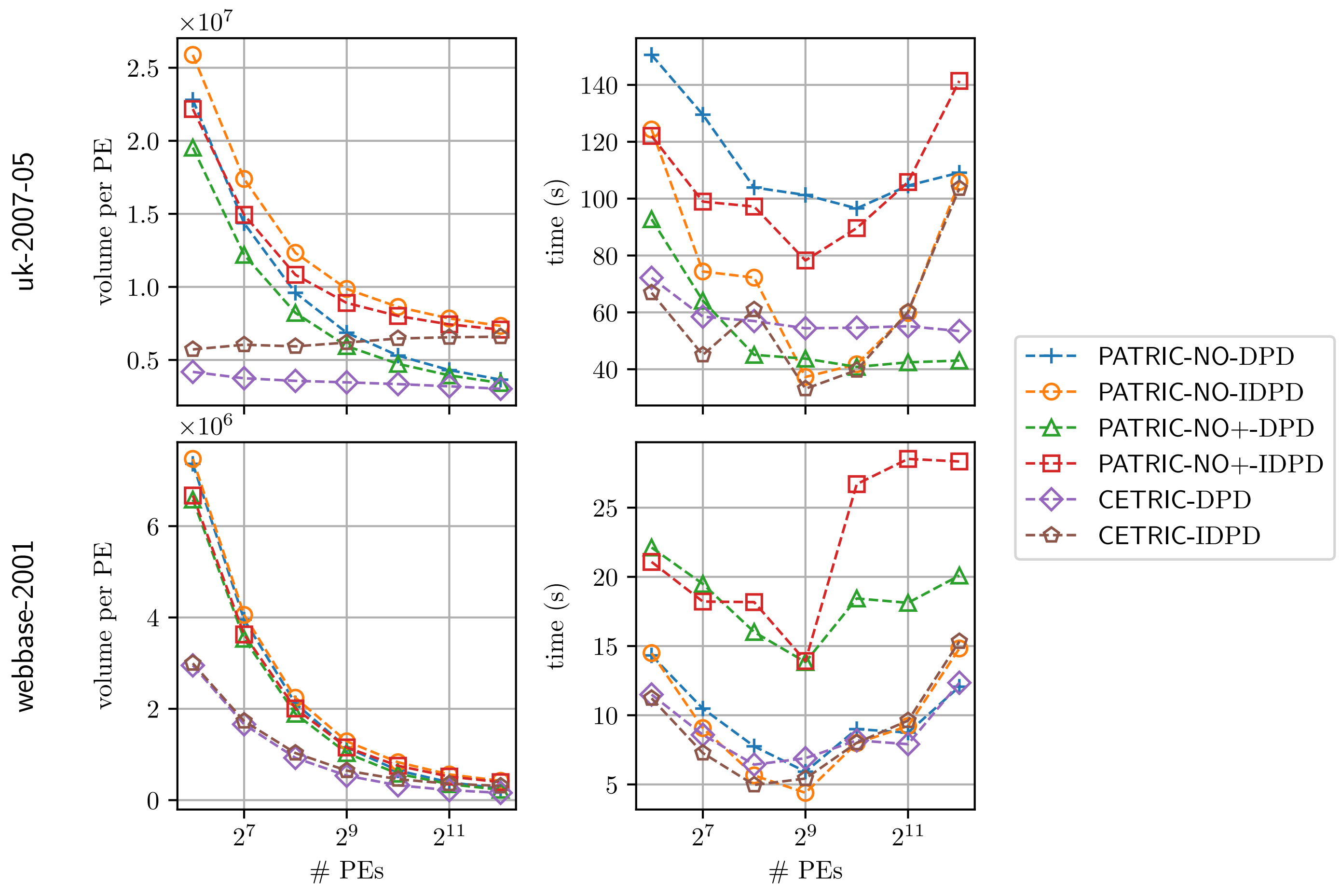


Figure 11: Strong scaling on web graphs using 64 to 4096 PEs on the SuperMUC-NG.

underestimates the actual cost required for processing the problematic vertex, it adds additional vertices to the processor P_i. Therefore the vertex has fewer ghost vertices as neighbors, which leads to less communication involving P_i. This means that DPD trades off imbalance in terms of local work for less communication. Due to network latency this results in better scalability. While DPD does not achieve a balance in terms of local work, it does not isolate the problematic vertex and the processor where it is located receives fewer messages which prevents a scaling bottleneck.

For 1024 to 4096 PEs PATRIC-NO+-DPD outperforms CETRIC-DPD. This is caused by the final Sparse-All-To-All operation (see Section 4.2.4) which takes longer for CETRIC than for PATRIC-NO+, despite CETRIC's lower communication volume, but we could not identify the cause of that.

Our results on webbase-2001 are similar. CETRIC reduces the message volume by around 50% on average, which results in outperforming PATRIC-NO in terms of running time. The algorithm variants using IDPD again perform better than those using DPD on up to 512 PEs, but then hit a scaling wall as already discussed for uk-2007-05. From 512 PEs onward, CETRIC-DPD is the best algorithm configuration. PATRIC-NO+ is outperformed by all other configurations.

The details of the balancing issue are shown in Figure 12. Again, a single PE causes a high imbalance. Opposed to uk-2007-05 the overloaded PE does not have a high amount of outgoing messages, but receives many messages.

As discussed before in Section 4.2.3 we experimented with several load balancing approaches which try to offload the work from the problematic processors, but they only resulted in a higher overall running time, without affecting the scalability.

Still, our experiments suggest that by combining the cost functions IDPD and DPD we can deal with the scaling issues we encountered. Figure 12 shows that if a processor has a send or receive volume which is by magnitudes higher than the average, the imbalance $\epsilon = \left|1 - \frac{C(V_i)}{\alpha}\right|$ is also high.

Because both cost functions are dependent on the out-degree of the neighbors of a vertex, we can compute both functions during load balancing without sacrificing running time. By default, using IDPD is desired, but if we detect a high imbalance ϵ for a single processor, we switch to DPD.

Road Networks We evaluate our algorithm on road networks of Europe and the USA. Road networks are very sparse, almost planar and have low average degree, as can also be seen in the degree distribution plots in Appendix A. We therefore expect these networks to have a low number of triangles and especially few Type-3 triangles. As our algorithm only requires communication for these triangles, we expect it to outperform PATRIC-NO. Figure 13 shows our results.

On europe we see that triangle counting is generally really fast due to the sparseness of the graph. PATRIC-NO takes only 0.3 seconds using 64 cores. We further notice that the overall communication volume is very low.

For example, live-journal and usa both require around 600-700MB of memory, but the required communication volume for live-journal using 1024 PEs is by a factor of 14 higher than for the similarly sized road network.

The communication reduction employed by PATRIC-NO+ already reduces the message volume by one third, but this does not pay off in terms of running time. Only CETRIC is able to significantly beat the competitors in terms of communication volume and running time.

On usa the major reduction in communication volume does not come from CETRIC's two-phase approach but from neighborhood reduction, as the resulting message volume is nearly the same

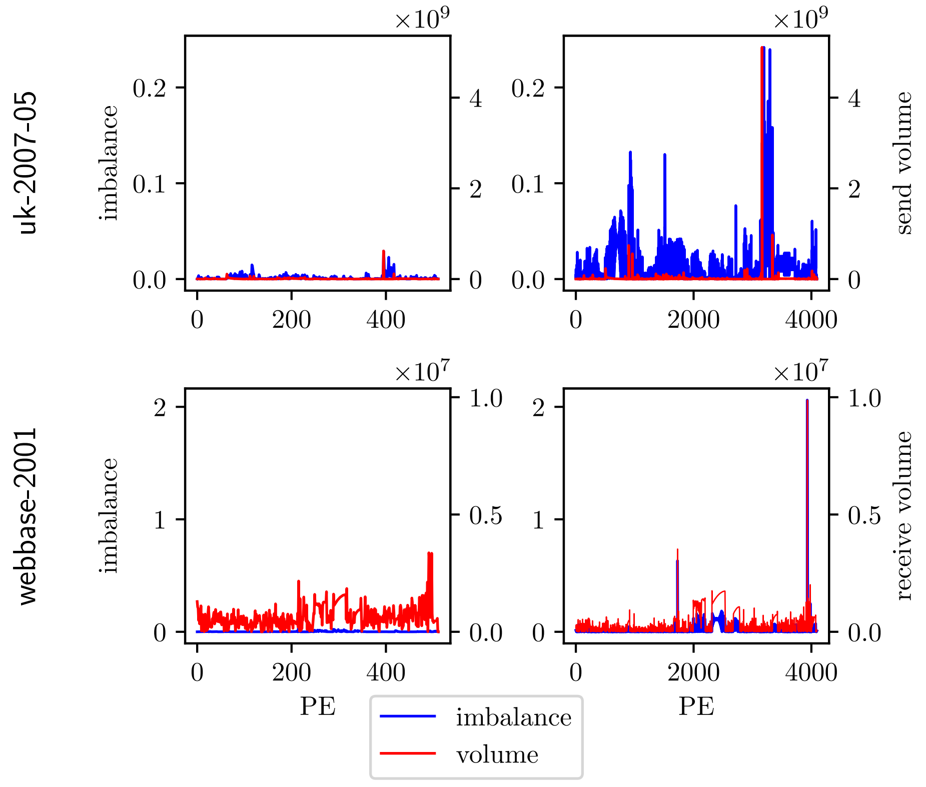


Figure 12: Load balancing imbalance and send / receive volume per PE of CETRIC-IDPD using 512 and 4096 cores on the SuperMUC-NG. The imbalance is defined as $\epsilon = \left|1 - \frac{C(V_i)}{\alpha}\right|$, where $\alpha = \frac{C(V)}{p}$.

for PATRIC-NO+ and CETRIC. As a result, the running time performance of all algorithms is nearly the same.

Road networks generally have small cuts, and together with the low degree there do not exist many vertices which can be removed from the transmitted neighborhoods by CETRIC.

We further notice that the choice of cost function has no impact on performance, neither for PATRIC-NO nor for CETRIC. Figure 14 shows the per processor time required for local work on europe. We see that it is evenly distributed for both cost functions. A reason for this is the degree distribution of the graph. As the degree is evenly distributed, the same holds for the required amount of local work per vertex and the estimated cost, which leads to an even work distribution for both cost functions. The same applies to usa.

Social Networks From the graph family of social networks we further investigate live-journal and orkut. In our previous experiments on the shared memory machine (see Figure 8) we noticed that CETRIC was able to greatly reduce the communication volume, but its impact reduced with increasing number of processor. Now considering our measurements for more than 64 PEs on the SuperMUC-NG in Figure 15, we see that CETRIC only slightly reduces the communication volume. Opposed to web graphs, using DPD as cost function yields a better running time.

Additionally, the indirection introduced by neighborhood reduction and Flag Intersection seems to slow down our algorithm on these instances, making it even slower than PATRIC-NO. We therefore disabled neighborhood reduction and Flag Intersection. This variant's (listed as CETRIC-plain) running time is almost identical to PATRIC-NO.

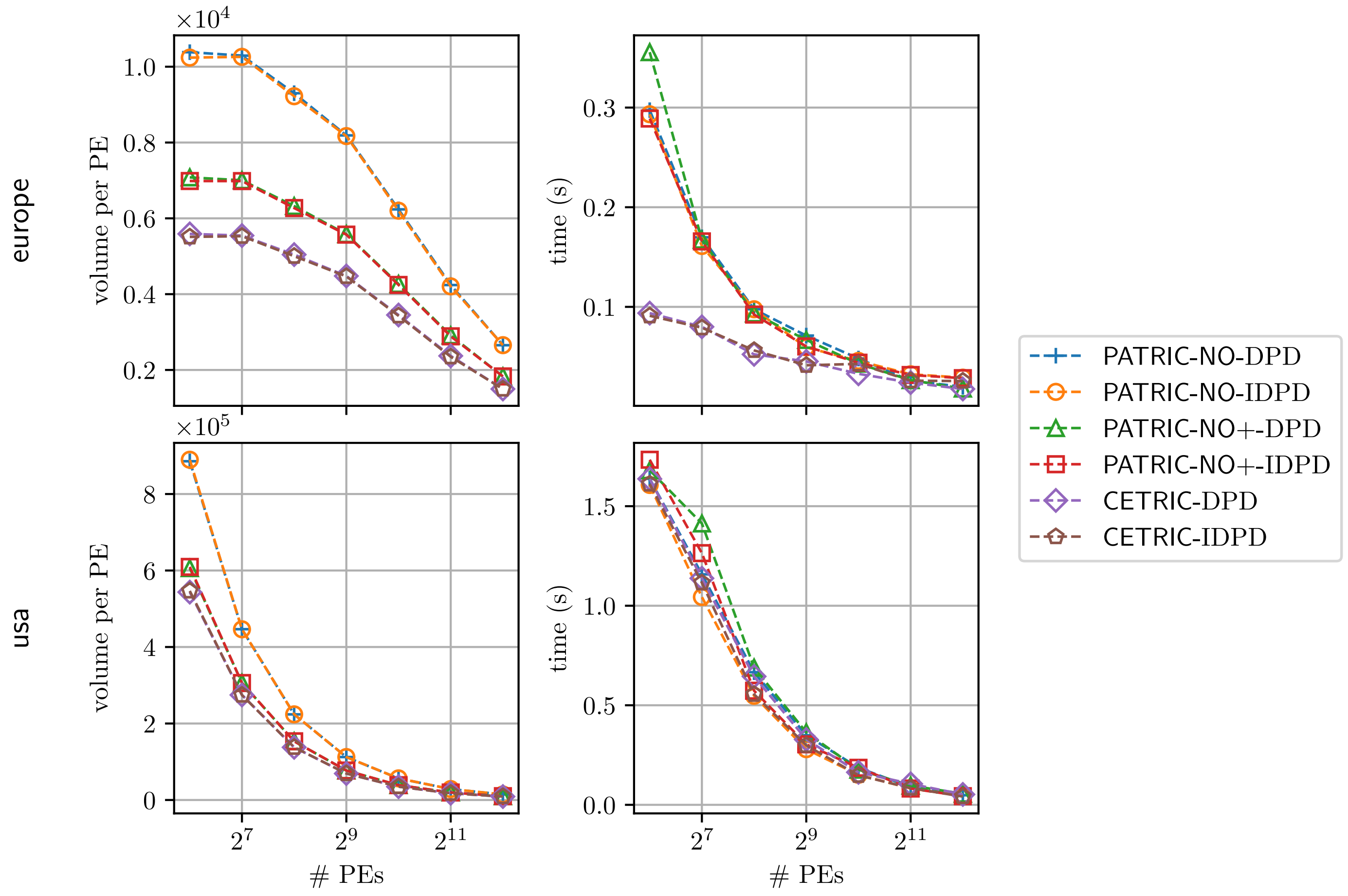


Figure 13: Strong scaling on road networks using 64 to 4096 PEs on the SuperMUC-NG.

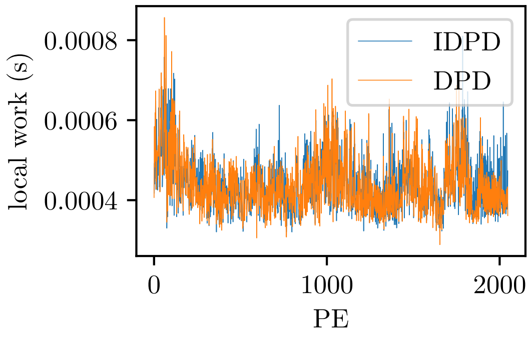


Figure 14: Local work of CETRIC on europe for different cost functions using 2048 PEs on the SuperMUC-NG.

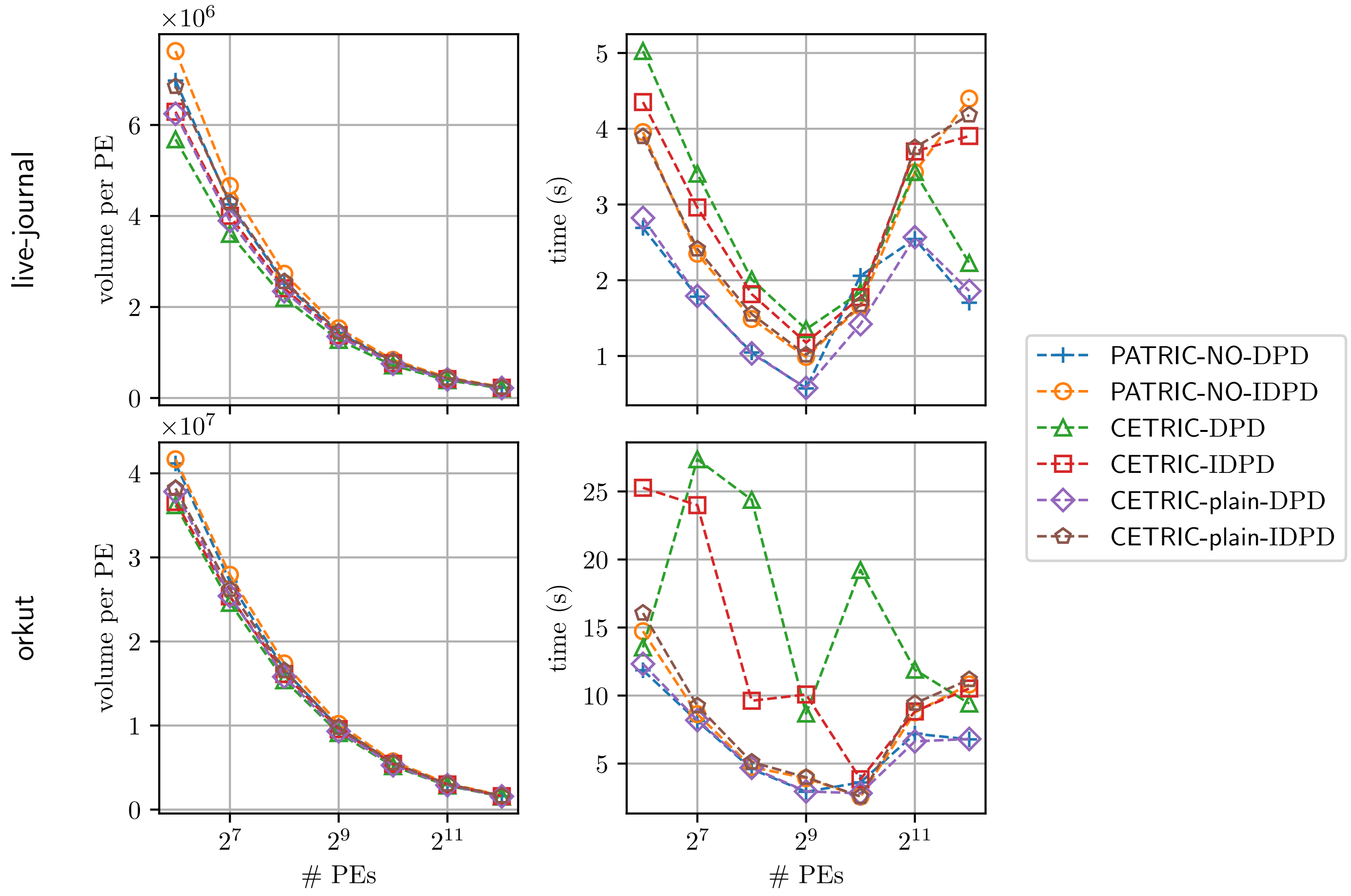


Figure 15: Strong scaling on social networks using 64 to 4096 PEs on the SuperMUC-NG.

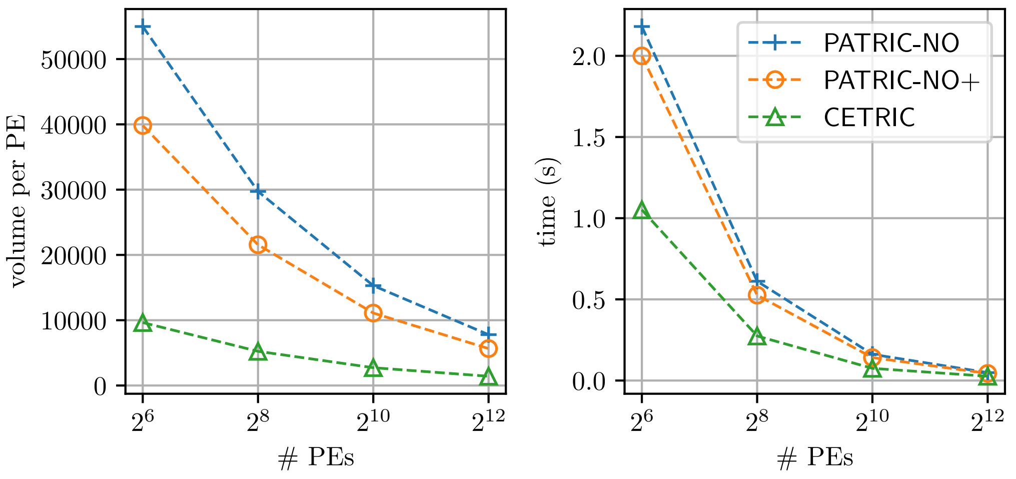


Figure 16: Strong scaling on RGG(2^{26}, 0.55) using 64 to 4096 PEs on the SuperMUC-NG.

The social instances have different structural properties than web graphs like uk-2007-05 and webbase-2001. Taking a closer look at the degree distributions in Appendix A, we see that all graphs have a skewed degree distribution and the web graphs are even larger. Still, the degree distribution for live-journal and orkut is more skewed, meaning that the graphs only have a very small number of high degree vertices, while the web graphs have more high degree vertices.

We therefore assume that our algorithm does not work well on graphs with a very skewed degree distribution.

Synthetic Instances For strong scaling experiments we use the RGG and RHG generators from [25]. As stated before, we do not employ load balancing but rely on the vertex partition provided by the distributed generator.

Figure 16 shows our results using the 2D Random Geometric Graph (RGG) model with $n = 2^{26}$ vertices and $r = 0.55\sqrt{\frac{\ln n}{n}}$. We see that CETRIC clearly outperforms PATRIC-NO and PATRIC-NO+. It reduces the communication volume by 70% and the running time by 48% on average compared to PATRIC-NO+. For 512 PEs the running time is even reduced by 60%.

CETRIC's good performance may be explained with the structure of the generated graphs. The RGG generator divides the 2D unit plane into equally sized cells. As vertices are adjacent if they lie in a circle with radius r, this leads to a very small number of Type-3 triangles. For example, using the given generator configuration using 4096 cores, the graph contains a total of 1.9 billion triangles, but only $144,000$ are Type-3 triangles. Therefore CETRIC may remove a large number of edges from the graph after the local phase and only has to send a fraction of the data PATRIC-NO requires. PATRIC-NO finds 25.4 million of the 1.9 billion triangles based on communicating with neighbors.

We also use the Random Hyperbolic Graph (RHG) model for evaluating CETRIC.We set $\overline{d} = 16$ and choose $\gamma \in \{3.0, 2.8\}$. For a lower value of γ the vertices become more concentrated at the center of the disk.

In Figure 17 we compare CETRIC to PATRIC-NO in terms of communication volume and running time. We generate instances with $n = 2^{26}$ and $n = 2^{28}$ vertices. For both values for γ and graph sizes, CETRIC reduces the communication volume by over 70%, regardless of the number of PEs. The running time is reduced by 40% on average, but with increasing number of PEs, CETRIC's lead in running time becomes lower.

For $\gamma = 3.0$ both CETRIC and PATRIC-NO scale well with increasing number of PEs, but

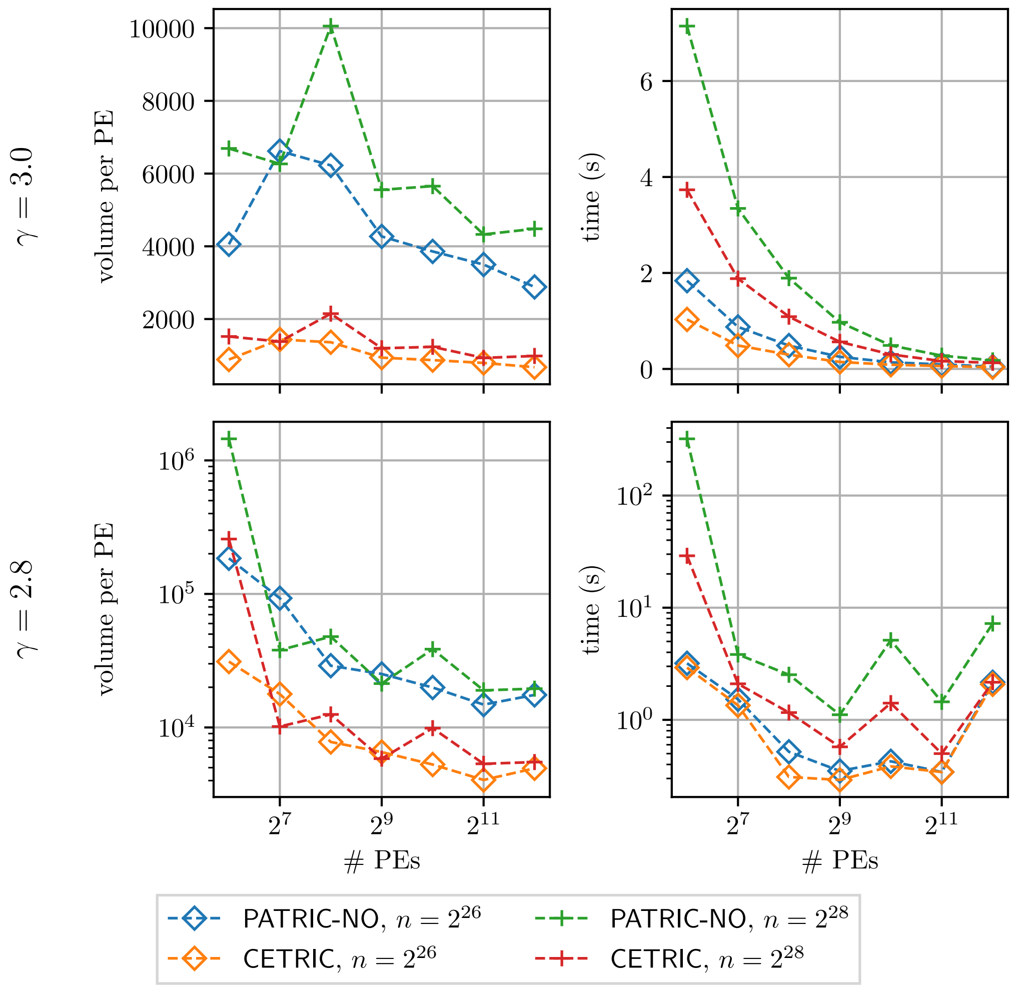


Figure 17: Strong scaling on RHG(n, 16, γ) using 64 to 4096 PEs on the SuperMUC-NG. Note that for $\gamma = 2.8$ we use a logarithmic scale for readability.

for $\gamma = 2.8$ the scalability of both algorithm becomes worse, but CETRIC still outperforms PATRIC-NO. We assume that the larger number of vertices around the center increases the cut size, which increases the communication volume. This also leads to very high commmunication volume for 64 PEs. For readability reasons, we use a logarithmic scale for $\gamma = 2.8$.

We omit PATRIC-NO+ from the results, as it performed worse than PATRIC-NO.

5.2.2. Weak Scaling

For our weak scaling experiments we use the distributed RGG and RHG generators from [25] introduced above.

Random Geometric Graphs For the RGG generator, we set the radius to $r = r_{\text{coeff}}\sqrt{\frac{\ln n}{n}}/\sqrt{p}$, where p is the number of PEs and $r_{\text{coeff}} \geq 0.55$. According to Funke et al. [25] this ensures that the graph is almost always connected and that the graph size per PE stays constant with increasing number of PEs. We noticed that for $r_{\text{coeff}} = 0.55$ the graph contains a very small number of triangles, which is why we set the radius slightly higher.

Figure 18 shows our result with for $\frac{n}{p} = 2^{20}$ vertices per PE. Under weak scaling, the communication volume decreases with the number of PEs, but the running time stays constant. We

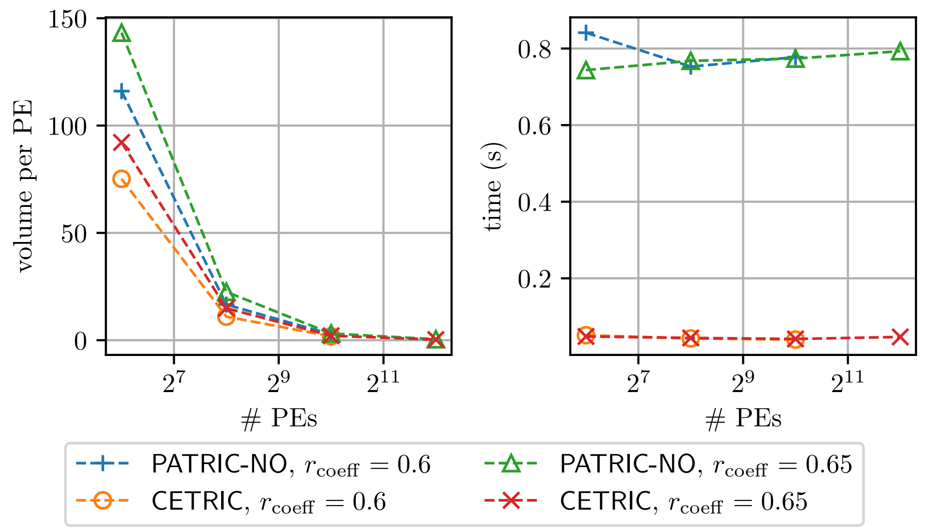


Figure 18: Weak scaling on RGG($2^{20} \cdot p$, r_{coeff}) using the SuperMUC-NG.

see that CETRIC reduces the communication volume by one third compared to PATRIC-NO and reduces the running time by 95%.

For $r_{\text{coeff}} = 0.6$ we do not report results for more than 1024 PEs. Due to memory limitations of the SuperMUC-NG we were not able to generate these instances in a reasonable time.

Random Hyperbolic Graphs For the RHG generator, we again set the number of vertices per PE to $\frac{n}{p} = 2^{20}$. As before we use the parameters from [25] and set $\overline{d} = 16$ and choose $\gamma \in \{3.0, 2.8\}$. Recall that for a lower value of γ the vertices become concentrated around the center.

Our results in Figure 19 show that CETRIC also scales better than PATRIC-NO in a weak scaling setting. For $\gamma = 3.0$ our algorithm reduces the required communication volume by 80% on average, the running time is reduced by 37%.

For $\gamma = 2.8$ our algorithm also clearly outperforms its competitor. The achieved volume reduction is 78%, but the reduction in running time is lower than for $\gamma = 3.0$, by 20% compared to PATRIC-NO. For 4096 PEs the running time is almost the same. As already mentioned for the strong scaling experiments, we assume that this is due to the larger cut of the graph.

R-MAT We also discuss the weak scaling behavior on R-MAT graphs. We use the parameterization from the Graph 500 benchmark as described above. Due to memory limitations of the used machine, we only used $\frac{n}{p} = 2^{14}$ vertices per PE. Due to these limitations we only show results for up to 2048 PEs, for the variants using DPD we could not report results for 2048 PEs due to memory constraints. Because there exists no R-MAT generator which prepartitions the graph, we use our load balancing mechanism for PATRIC-NO and CETRIC.

The results in Figure 20 show that both PATRIC-NO and CETRIC do not scale well on R-MAT graphs and CETRIC only achieves a slight reduction in communication volume. As already observed for the social instances in Section 5.2.1, the additional indirection introduced by neighborhood reduction and Flag Intersection slows down CETRIC.

We therefore also include CETRIC-plain. This algorithm configuration achieves a similar running time compared to PATRIC-NO but does not benefit from neighborhood reduction. IDPD as cost functions yields the best results.

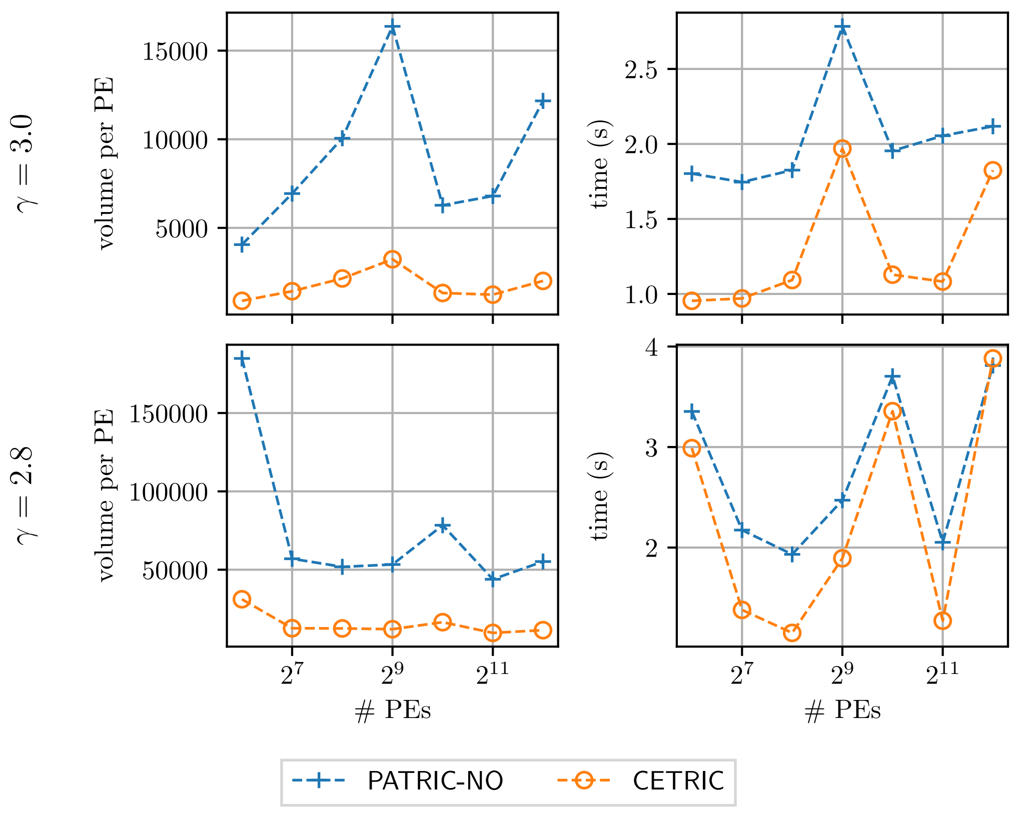


Figure 19: Weak scaling on RHG($2^{20} \cdot p$, 16, γ) using the SuperMUC-NG.

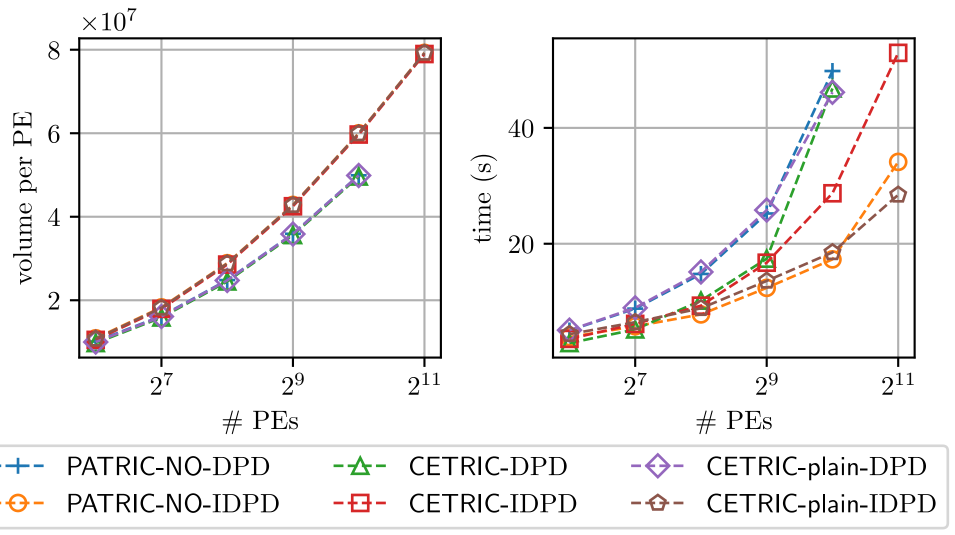


Figure 20: Weak scaling on R-MAT graphs with $\frac{n}{p} = 2^{14}$ using the SuperMUC-NG.

R-MAT graphs have a very skewed degree distribution. The experiments on social instances in Section 5.2.1 already showed that CETRIC and PATRIC-NO do not scale well on these instances and that CETRIC only slightly reduces the communication volume. We experience the same behavior on the synthetic instances.

5.3. Conclusion

Our experiments have shown that CETRIC reduces the communication volume of PATRIC by almost 50% on average on real-world instances, which results in a reduced running time by one third on average, especially when using up to 512 PEs.

Our algorithm works particularly well on web graphs. They have a skewed degree distribution but still contain many high degree vertices and have a high locality, which allows our algorithm to greatly reduce the communication volume.

On graphs representing social networks our algorithm reduces the communication volume by an average of 40%, and reduces the running time by the same amount when using up to 64 PEs, but with increasing number of processor it does not improve much upon PATRIC-NO, because with increasing cut size our algorithm only slightly reduces the communication volume compared to the competitor.

CETRIC clearly outperform PATRIC-NO on synthetic graphs with high locality and small cuts, as this allows for a communication reduction of 70%.

While we achieve the lowest running times using IDPD as cost function, we uncover scaling issues related to IDPD using more than 512 PEs. It leads to isolated vertices which cause a single PE having to send and receive many messages, which in turn results in a communication bottleneck.

These imbalances may be detected by measuring load imbalance with respect to the cost function. While DPD does not work as well as IDPD on a smaller number of PEs, it is less sensitive to imbalance. By detecting imbalances and balancing load according to DPD if they occur, we can tame the scalability issues.

While our algorithm scales better than PATRIC-NO on Random Geometric Graphs and Random Hyperbolic Graphs, it does not give any benefit over the competitor in scaling experiments on R-MAT graphs, where both algorithm show poor weak scaling behavior.

6. Message Compression with Bloom Filters

Bloom filters may be used to represent sets in a compressed form and allow for membership queries with a certain false positive rate.

This makes them suitable for lowering the required communication volume of our algorithm. We describe two variants of CETRIC which use Bloom filters, an exact algorithm and an approximation algorithm.

In Section 6.1 we give an introduction to Bloom filter and Section 6.2 describes our extensions of CETRIC which incorporate Bloom filters.

6.1. Introduction to Bloom Filters

A Bloom filter is a space-efficient probabilistic set data structure that supports membership queries which has been first introduced by Burton H. Bloom [11]. The membership query using a Bloom filter allows for false positives, i.e. if the query returns true for an element x this may be wrong with a certain probability, but if the query returns false we can be sure that the element is not contained in the set.

Using a Bloom filter, a set $S \coloneqq \{x_1, \dots, x_n\}$ of n elements may be represented using a bit-vector $\mathtt{b}$ of m bits and the data structure uses k independent hash functions $h_1, \dots, h_k$ which map to the range $\{0, \dots, m-1\}$. Initially, all bits in $\mathtt{b}$ are set to zero. If an element x is inserted into the filter, $h_i(x)$ is evaluated and the corresponding bit $\mathtt{b}[h_i(x)]$ is set to one for each $i = 1, \dots, k$.

To check if an element x is contained in S, we evaluate each hash function. If $\mathtt{b}[h_i(x)] = 1$ for each $i = 1, \dots, k$, we assume that $x \in S$, if not $x \notin S$.

Under the assumption that all hash functions are perfectly random, the false positive rate may be determined. We assume that all elements have been inserted into the Bloom filter. Then according to Broder and Mitzenmacher [15] the false positive rate q is given by

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Given the size n of the set and the size m of the bit-vector, the optimal number of hash functions $k_{\text{opt}} = \frac{m}{n} \ln 2$ minimizes the false positive probability, to $\left(\frac{1}{2}\right)^{k_{\text{opt}}}$. To achieve a fixed false positive rate q, we can therefore set k to k_{opt} and m to

$$m = -\frac{n \ln q}{(\ln 2)^2}$$

A number of Bloom Filter variants exist. Counting Bloom filters allow for element deletion using a small counter for each hash value. Mitzenmacher [37] introduces compressed Bloom filters, which are Bloom filters optimized for compression. Applications of Bloom filters include speeding up join operations in databases [15] and distributed web caching [24].

For a more detailed overview of Bloom filters we refer the reader to Tarkoma et al.'s survey paper [53].

6.2. Combining CETRIC with Bloom Filters

The compact representation of Bloom filters predestines them for applications in distributed memory algorithms, because they allow for compression of messages.

The integration of Bloom filters into CETRIC to reduce message size is straightforward. Instead of sending neighborhoods as vertex sets, we represent the neighborhoods by using Bloom filters. When P_i has to send N_v^+ of a vertex v to processor P_j, it constructs a Bloom Filter of size m using k hash functions. We discuss later how these parameters are chosen. P_i then inserts each element in N_v^+ into the Bloom filter and sends the resulting filter to processor P_j. We see that these modifications only impact the global phase of CETRIC, the local phase remains unchanged.

6.2.1. Exact Algorithm

Upon receiving a neighborhood N_v^+ represented as a Bloom filter P_j handles it the same way as in CETRIC, by iterating over each vertex $u \in N_v^+ \cap V_j$ (recall that this partial neighborhood is locally available to P_j) and checking for each $w \in N_u^+$ whether w is contained in the Bloom filter. Opposed to plain CETRIC, we can now not be sure whether the triangle $\{v, u, w\}$ exists, because w may be a false positive. Only processor P_i knows if actually $w \in N_v^+$.

Therefore we need an additional algorithm phase, which filters false positives from the potential triangles. We call this the *filter phase.*

This can be done by collecting all potential triangles and sending them to the processor with index $\text{rank}(v)$. This processor then checks if the edge (v, w) exists and counts the triangle. Using this approach, the filter phase requires a communication volume of at least $3 \cdot T^{(3)}$, where $T^{(3)}$ is the number of Type-3 triangles in the graph.

If we only want to know the number of triangles, it suffices to request the edge query for (v, w) together with the number of potential triangles dependent on the existence of the edge to P_i.

Bloom Filter Parameterization Recall that the false positive rate q of a Bloom filter of size m using k hash functions which stores a neighborhood of size n is given by

$$q = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

If the main goal is to save communication during the global phase, we can set m to a constant fixed size and choose $k = k_{\text{opt}} = \frac{m}{n} \ln 2$ as this minimizes the false positive rate. Each neighborhood may then be represented using constant space m. Note that with increasing size of the neighborhood, the false positive rate also increases, which in turn leads to an increasing communication volume.

It may therefore be favorable to use a fixed false positive rate q instead, choose k optimally as above and then choose $m = -\frac{n \ln q}{(\ln 2)^2}$. The communication volume of the filter phase is then expected to be $3 \cdot (1 + q)T^{(3)}$. Further experimental evaluation is required to find an optimal parameter set.

Preliminary Results Due to time constraints, we were not able to evaluate the exact algorithm in practice, but preliminary experiments have shown that this approach is only feasible for graphs with a low number of Type-3 triangles. Consider the first column of Figure 21. We see that on webbase-2001 the lower bound on communication required for the filter phase, $3 \cdot T^{(3)}$, exceeds the communication volume of CETRIC's global phase by a magnitude. This suggests that an algorithm with an additional filter phase would be much slower than CETRIC. We observed similar results for the other graph instances.

Only on road networks such as europe the number of Type-3 triangles is low and we expect an implementation of the Bloom filter algorithm to outperform CETRIC.

6.2.2. Approximation Algorithm

If we are willing to sacrifice an exact result for reduction of communication volume, we can skip the communication intensive filter phase and instead exploit the probabilistic features of Bloom filters.

If we choose a fixed false positive rate q and choose the Bloom filter parameters m and k accordingly for each message, we can use CETRIC together with Bloom filters as an approximation algorithm, which counts all Type-1 and Type-2 triangles exactly and gives an approximation of the number of Type-3 triangles.

Again the main idea is straightforward. We use the Bloom filter algorithm from above. From the correctness of CETRIC's local phase follows that the number of Type-1 and Type-2 triangles, $T^{(1)}$ and $T^{(2)}$ is exactly determined by the local phase, i.e. $T_{\text{local}} = T^{(1)} + T^{(2)}$. The global phase counts T_{global} triangles, but each of these triangles is a false positive with probability q. The expected number of false positives is therefore qT_{local}. This gives an approximated total number of triangles $T_{\text{approx}} = T_{\text{local}} + (1 - q)T_{\text{global}}$.

The algorithm avoids the cost-intensive filter phase and is especially useful if the correctness of the result is mostly important for Type-1 and Type-2 triangles.

To achieve a fixed false positive rate q the size of the Bloom filters may not be constant, but instead we chose k and m to minimize the false positive rate with respect to the message size using the formula given above. This also gives us a minimal size m of the Bloom filter to achieve the desired false positive rate and therefore minimizes the communication volume.

Preliminary Results Due to time constraints we do not evaluate the approximative approach in depth. Our implementation uses `boost::dynamic_bitset` to represent Bloom filters and SpookyHash[5] as a hash function.

We show our preliminary results in Figure 21. We use CETRIC without Bloom filters and the variant with Bloom filters using the optimal number of hash functions k_{opt} and the optimal size m (BLOOM-k_{opt}) and 2 hash functions (BLOOM-2) for a fixed false positive rate of 1% as algorithm configurations. In the first column we report the total communication volume of the global phase, in the second column the running time of the algorithm and the relative error for the number of triangles $\epsilon = \left|1 - \frac{T_{\text{approx}}}{T}\right|$ in the right column.

We see that by using Bloom filters the communication volume is reduced. On webbase-2001 using k_{opt} the volume is reduced even further, but this does not pay off in terms of running time. Both configurations with Bloom filters are slower than CETRIC without delivering an exact result. Constructing the Bloom filters takes a lot longer than building message buffer without using compression and the achieved reduction in communication volume does not pay off. Using $k = 2$ hash functions the running time is lower and from 128 PEs onward the relative error stays almost constant, while for $k = k_{\text{opt}}$ it even exceeds the false positive rate of 1%.

On europe the Bloom filter algorithms are faster than CETRIC and using $k = k_{\text{opt}}$ the relative error stays low. Especially for small numbers of PEs, when the graph also has very few Type-3 triangles the error is low.

[5]https://burtleburtle.net/bob/hash/spooky.html

Our results suggest that the Bloom filter approach only works well on graphs with a low number of Type-3 triangles, where it also gives good approximations. We still expect that careful tuning of the Bloom filter construction may make the algorithm usable on other instances.

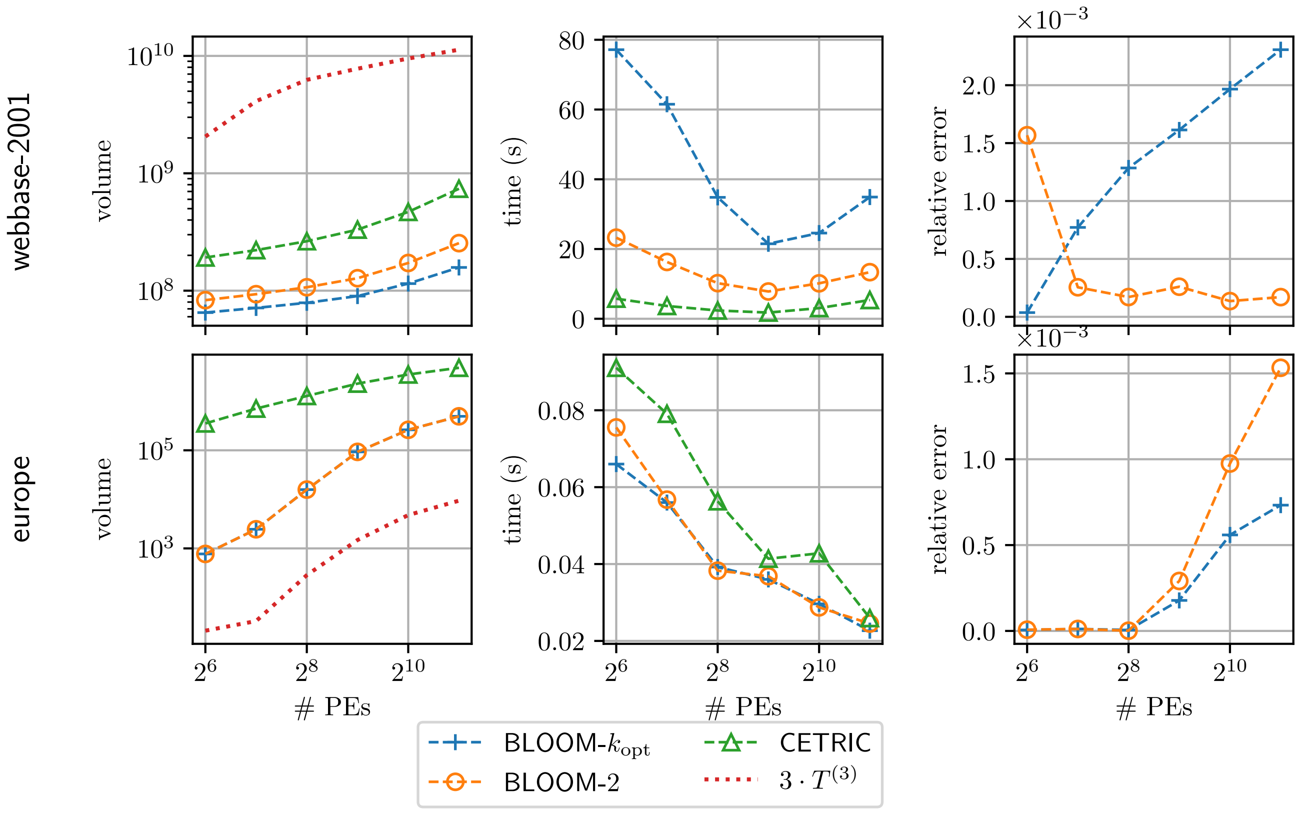


Figure 21: Preliminary experiments with Bloom filters on the SuperMUC-NG. The left column shows the total communication volume of the global phase, the middle column the running time and right column the relative error of the approximation $\epsilon = \left|1 - \frac{T_{\text{approx}}}{T}\right|$ for a false positive rate $q = 0.01$.

7. Conclusion

In this thesis, we proposed CETRIC, a message-passing-based parallel triangle counting algorithm designed for distributed memory systems. It builds on the algorithm by Arifuzzaman et al. [5] which is the only message-passing based distributed memory triangle counting algorithm to our knowledge.

We are able to reduce the amount of data which is sent between processors compared to their approach by exploiting locally available information. More precisely, our algorithm only requires communication for counting triangles where each endpoint is located on a different processor. We therefore can reduce the input after counting all local triangles and execute the communication-intensive distributed triangle counting procedure only on the graph consisting of cut edges. In addition, we suggest several other improvement techniques for faster parallel triangle counting. This includes faster set intersection by using bit-vectors and message buffering.

We show that our technique reduces the communication volume in theory and practice. We experimentally evaluated our algorithm by comparing its performance to Arifuzzaman et al.'s algorithm using up to 4096 PEs of the SuperMUC-NG and a large variety of real-world and synthetic graph instances. Previously, the competitor has only been tested using up to 1000 PEs and smaller input instances.

The experiments show, that our approach is able to reduce the communication required for exchanging neighborhoods for triangle counting by almost 50% on average on real-world instances, which results in a running time reduced by one third. Our algorithm outperforms the competitor on most real-world instances. On synthetic instances generated using the RGG and RHG model our algorithm performs even better due to the high locality and small cut size of the generated graphs. In general, our algorithm seems to work especially well on real-world networks which have a skewed degree distribution but still have many high degree vertices. In these cases the load balancing scheme of our algorithm is able to redistribute work evenly and we can greatly reduce the communication volume. In addition, our algorithm works well if the input is distributed among processors such that the cut is small, as this allows for reducing the communication volume even further. On social networks, our algorithm especially outperforms the competitor when using up to 64 PEs, but still reduces communication volume and running time with higher number of processors.

Additionally, we uncover scaling issues of the competitor algorithm which also apply to our algorithm. On instances with a skewed degree distribution the 1D partitioning scheme used in both algorithms leads to imbalances, which hinder scalability with increasing number of processors. We provide a heuristic which relies on ideas from Arifuzzaman et al.'s load balancing that allows to tame the imbalances.

We also suggest two variants of our algorithm, which use Bloom filters to represent neighborhoods in a compressed form. This allows to reduce the communication in the main phase of the algorithm, but comes at the cost of false positives. Therefore additional communication is required to filter false positives, which only pays off if the graph has a low number of triangles which are located on three distinct PEs.

For other graph instances, Bloom filters become useful if we discard the requirement of exactly counting triangles. Parameterizing the Bloom filters such that they achieve a fixed false positive rate, we can approximate the number of cut triangles, while still counting other triangles exactly and skipping the false positive filtering. Due to time constraints we do not provide a fine-tuned implementation of these variants, but based on preliminary results we expect a good performance for a careful implementation.

Future Work As already suggested, focusing on implementation details of CETRIC combined with Bloom filters promises good results.

There also exist further directions for optimizing CETRIC and providing a more detailed analysis.

Instead of interleaving communication and local work, our algorithm works in two phases. It will be interesting to investigate how this improves cache locality and which role it plays in making CETRIC fast.

Our current implementation uses point-to-point communication, which makes sense if each PE only has a limited number of communication partners. If the graph contains vertices which are connected to many other PEs, other communication schemes may be favorable.

The major achievement of CETRIC is that it reduces the required communication volume, but preliminary experiments on neighborhood reductions suggest that this does not always lead to a reduction in running time. We assume that due to the high communication bandwidth of the SuperMUC-NG saving communication does not always directly translate to running time reduction. We expect our algorithm to be even faster compared to PATRIC-NO on a machine with lower communication speed, because the running time is dominated even more by communication.

Our experiments have shown that the load balancing scheme still leaves room for improvement. On the one hand, it may be interesting to investigate additional alternative cost functions which aim at minimizing communication volume. On the other hand, we see that 1D partitioning hits a scaling wall on graph instances with a skewed degree distribution. Therefore revisiting 2D partitioning may be necessary. Another alternative approach would be the use of dynamic load balancing instead of static load balancing.

A. Degree Distribution of Real-World Instances

In Figure 22 we visualize the degree distribution of the real-world instances used in our experiments, summarized in Table 2. For each instance we plot the frequency of each degree value in the graph interpreted as an undirected input. We further show the distribution of the out-degree and in-degree of each instance after the graph has been oriented according to the degree-based total ordering $\prec$ introduced in Section 3.1.

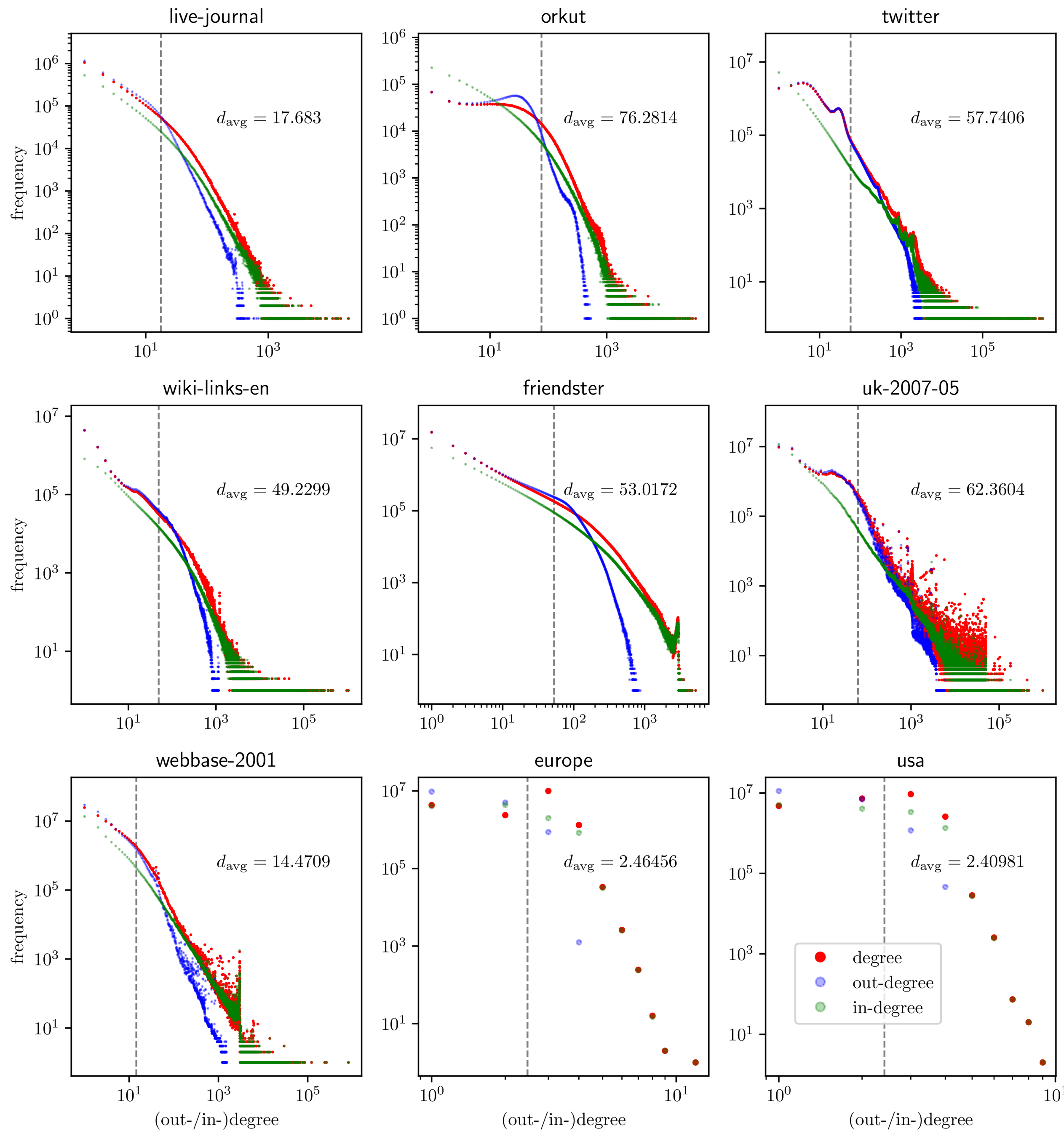


Figure 22: Degree distribution of real-world instances listed in Table 2. We consider the input graph as undirected and define the out- and in-degree with respect to the degree-based total ordering $\prec$ introduced in Section 3.1. The dashed vertical line indicates the average degree d_{avg}.

B. Detailed Timing for Algorithm Phases

In this section we show the running time for the phases of each algorithm configuration. For CETRIC, we split the algorithm core in the local and global phase, while PATRIC uses a single phase. We report the results for the web, road and social graph instances discussed in detail in Section 5.2.1 in Figure 23. For live-journal and orkut we use CETRIC without neighborhood reduction and Flag Intersection based on the result from Section 5.2.1.

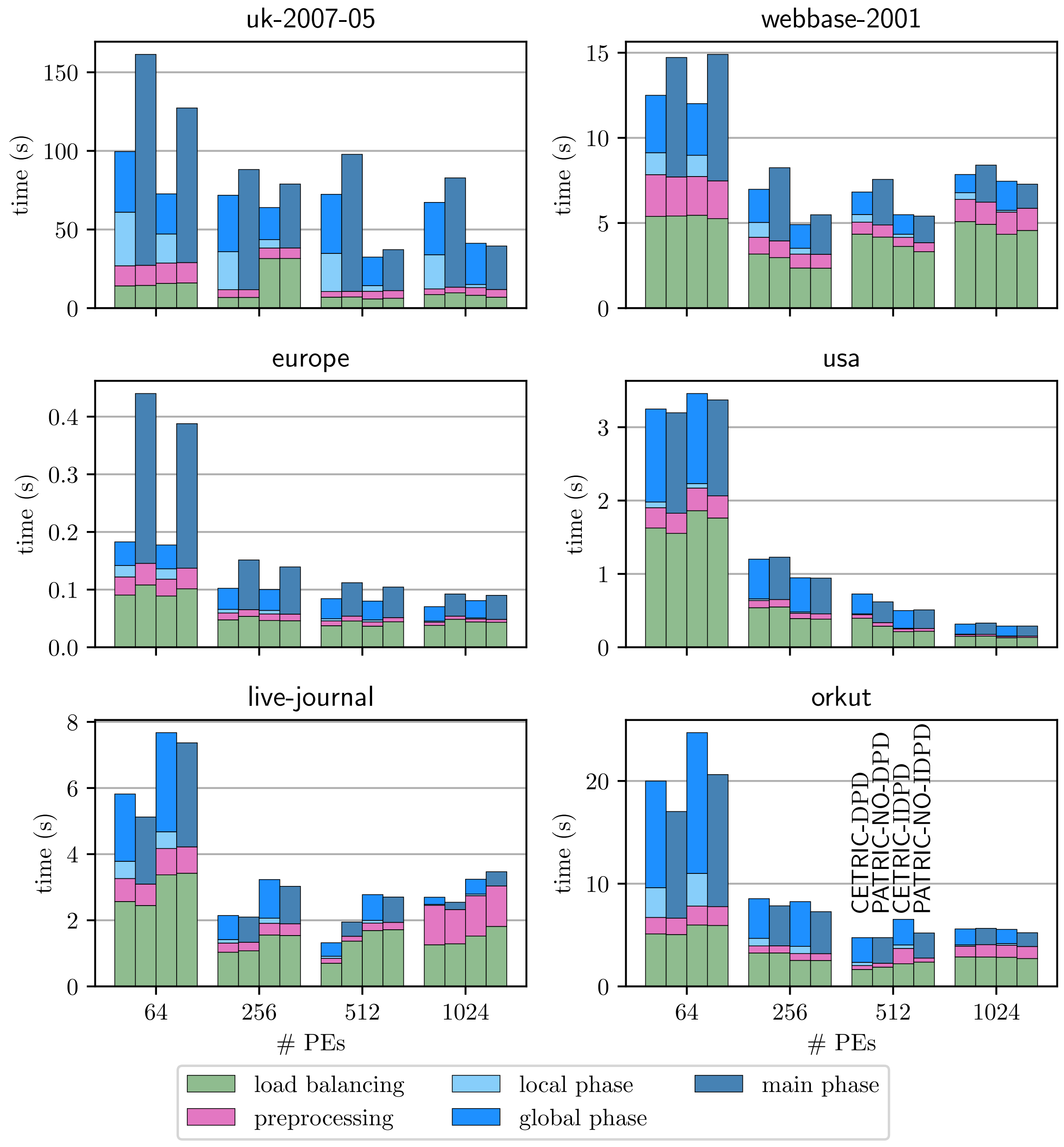


Figure 23: Running time of the algorithm phases. The configurations from left to right are CETRIC-DPD, PATRIC-NO-DPD, CETRIC-IDPD and PATRIC-NO-IDPD.

C. Kernelization-based Approximative Sequential Triangle Counting

At an early stage of this work, we considered using kernelization for approximative triangle counting. Kernelization relies on reducing the input problem until a small, easily solvable subproblems, the *kernel* remains. While preliminary experiments suggest that the algorithm is slower than other approximation algorithms in practice, we still provide some insights on our algorithm here for reference.

The key idea is to iteratively remove vertices with degree 0, 1 and 2 from the input and count the triangles incident on such vertices. If all low degree vertices have been removed, we apply the coloring step from the approximation algorithm by Pagh and Tsourakakis [42] (see Section 3.3) and only keep monochromatic edges.

From the resulting graph, we again remove all low degree vertices and repeat this procedure until no vertex is left.

The algorithm may be implemented using a priority queue of all vertices Q order by increasing degree. Pseudocode is listed in Algorithm 8.

Algorithm 8: Color-and-Reduce

```
$T \leftarrow 0$
$\mathsf{Q} \leftarrow \{v \in V, \text{sorted by increasing degree}\}$
$\mathsf{round} \leftarrow 0$
while $Q \neq \emptyset$ do
    $v \leftarrow$ Q.popMin()
    if $d_v = 1$ then
        $\{u\} \leftarrow N_v$
        Q.decreaseKey($u$)
        $G \leftarrow G - v$
    else if $d_v = 2$ then
        $\{u, w\} \leftarrow N_v$
        if $\{u, w\} \in E$ then
            $T \leftarrow T + N^{2*\mathsf{round}}$
        Q.decreaseKey($u$)
        Q.decreaseKey($w$)
        $G \leftarrow G - v$
    else
        for $v \in V$ do
            $c(v) \leftarrow \text{random}(\{0, 1\})$
            $d'_v \leftarrow d_v$
        $E \leftarrow \{\{u, v\} \in E \mid c(u) = c(v)\}$
        for $v \in V$ do
            while $d'_v < d_v$ do
                Q.decreaseKey($u$)
        $\mathsf{round} \leftarrow \mathsf{round} + 1$
```

References

[1] L. Akoglu and B. Dalvi. Structure, tie persistence and event detection in large phone and sms networks. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 10–17, 2010.

[2] M. Al Hasan and V. S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.

[3] M. Alam and M. Khan. Parallel algorithms for generating random networks with given degree sequences. *International Journal of Parallel Programming*, 45(1):109–127, 2017.

[4] M. J. Appel and R. P. Russo. The connectivity of a graph on uniform points on $[0, 1]^d$. *Statistics & Probability Letters*, 60(4):351–357, 2002.

[5] S. Arifuzzaman, M. Khan, and M. Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, pages 529–538, 2013.

[6] S. Arifuzzaman, M. Khan, and M. Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 527–534, 2015.

[7] S. Arifuzzaman, M. Khan, and M. Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(1):1–34, 2019.

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

[9] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 623–632, 2002.

[10] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 16–24, 2008.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 587–596, 2011.

[13] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the Thirteenth International International Conference on World Wide Web*, WWW '04, pages 595–601, 2004.

[14] U. Brandes. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[15] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[16] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[17] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[18] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 672–680, 2011.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 10, 2004.

[20] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 2009.

[21] R. Diestel. Extremal graph theory. In *Graph Theory*, pages 173–207. Springer, 2017.

[22] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the National Academy of Sciences*, 99(9):5825–5829, 2002.

[23] S. Eubank, V. A. Kumar, M. V. Marathe, A. Srinivasan, and N. Wang. Structural and algorithmic aspects of massive social networks. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 718–727, 2004.

[24] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[25] D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. von Looz. Communication-free massively distributed graph generation. *Journal of Parallel and Distributed Computing*, 131:200–217, 2019.

[26] O. Green, P. Yalamanchili, and L.-M. Munguía. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2014.

[27] L. Hoang, V. Jatala, X. Chen, U. Agarwal, R. Dathathri, G. Gill, and K. Pingali. DistTC: High performance distributed triangle counting. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.

[28] T. Hoefler and J. L. Traff. Sparse collective operations for mpi. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[29] Y. Hu, H. Liu, and H. H. Huang. Tricore: Parallel triangle counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[30] L. Hübschle-Schneider and P. Sanders. Linear work generation of R-MAT graphs. *Network Science*, 2020.

[31] M. Jha, C. Seshadhri, and A. Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 589–597, 2013.

[32] J. Kunegis. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, 2013.

[33] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

[34] Leibniz Supercomputing Centre. Supermuc petascale system. https://www.lrz.de/services/compute/museum/supermuc/systemdescription/.

[35] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[36] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna. Web data commons - hyperlink graph. http://km.aifb.kit.edu/sites/webdatacommons/hyperlinkgraph/index.html.

[37] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.

[38] M. E. J. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[39] M. E. J. Newman. *Networks*. Oxford university press, 2018.

[40] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *Proceedings of the national academy of sciences*, 99(suppl 1):2566–2572, 2002.

[41] M. Ortmann and U. Brandes. Triangle listing algorithms: Back from the diversion. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–8. SIAM, 2014.

[42] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[43] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh. Mapreduce triangle enumeration with guarantees. In *Proceedings of the 23rd ACM International Conference on Information & Knowledge Management*, pages 1739–1748, 2014.

[44] R. Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–4, 2017.

[45] M. Rahman and M. Al Hasan. Approximate triangle counting algorithms on multi-cores. In *2013 IEEE International Conference on Big Data*, pages 127–133. IEEE, 2013.

[46] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, et al. Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2017.

[47] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

[48] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment*, 6(4):277–288, 2013.

[49] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. PhD thesis, Universität Karlsruhe (TH), 2007.

[50] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, 2015.

[51] Statista. Number of monthly active facebook users worldwide as of 4th quarter 2020. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/, January 2021.

[52] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international Conference on World Wide Web*, WWW '11, pages 607–614, 2011.

[53] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[54] The Graph 500 steering commitee. The Graph 500 benchmark. https://graph500.org.

[55] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis. Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[56] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, 2011.

[57] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 837–846, 2009.

[58] L. Wang, Y. Wang, C. Yang, and J. D. Owens. A comparative study on exact triangle counting algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8, 2016.

[59] D. J. Watts and S. H. Strogatz. Collective dynamics of “small-world” networks. *Nature*, 393(6684):440–442, 1998.