

Asynchronous Distributed-Memory Triangle Counting and LCC with RMA Caching

András Strausz*, Flavio Vella†, Salvatore Di Girolamo‡, Maciej Besta‡ and Torsten Hoefer‡

*‡Dept. of Computer Science, ETH Zürich, Zürich, Switzerland

†Dept. of Engineering and Computer Science, University of Trento, Trento, Italy

*strausza@student.ethz.ch, †flavio.vella@unitn.it, ‡firstname.lastname@inf.ethz.ch

Abstract—Triangle count and local clustering coefficient are two core metrics for graph analysis. They find broad application in analyses such as community detection and link recommendation. To cope with the computational and memory demands that stem from the size of today’s graph datasets, distributed-memory algorithms have to be developed. Current state-of-the-art solutions suffer from synchronization overheads or expensive pre-computations needed to distribute the graph, achieving limited scaling capabilities. We propose a fully asynchronous implementation for triangle counting and local clustering coefficient based on 1D partitioning, using remote memory accesses for transferring data and avoid synchronization. Additionally, we show how these algorithms present data reuse on remote memory accesses and how the overall communication time can be improved by caching these accesses. Finally, we extend CLaMPI, a software-layer caching system for MPI RMA, to include application-specific scores for cached entries and influence the eviction procedure to improve caching efficiency. Our results show improvements on shared memory, and we achieve 14x speedup from 4 to 64 nodes for the LiveJournal1 graph on distributed memory. Moreover, we demonstrate how caching remote accesses reduces total running time by up to 73% with respect to a non-cached version. Finally, we compare our implementation to TriC, the 2020 graph champion paper, and achieve up to 100x faster results for scale-free graphs.

Index Terms—asynchronous computing, caching, distributed computing, local clustering coefficient, RDMA, triangle counting

I. INTRODUCTION

Complex real-world systems can be modeled, analyzed, and optimized through their respective graph representations. These systems range from social networks [1], to biological networks [2], to the full Internet [3]. Data mining, information retrieval, recommendation systems, and fraud detection are just a few applications of graph analysis [4]–[8].

The local clustering coefficient (LCC) [9] indicates the likelihood that neighbors of a node are also neighbors to each other, which has applications in the link prediction problem [10]. In particular, this metric has been proven useful in many applications, such as community detection [11], [12] or link recommendation [13]. In the former case, LCC is used to detect communities in, e.g., social networks, distinguishing between vertices that are central to the cluster from others on its frontier. In the latter, clustering coefficient is used to locate thematic relationships by looking at the graph of hyperlinks.

The LCC of a vertex is given as the fraction of the pairs of its neighbors that are themselves connected by an edge. Figure 1 (left) shows an example of LCC scores on a toy graph.

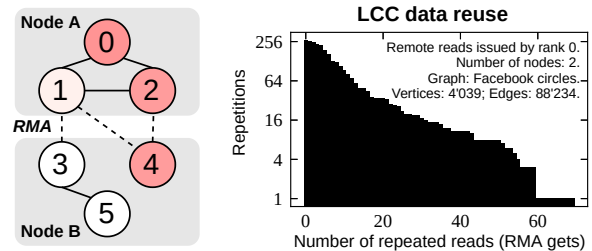


Fig. 1: LCC example and data reuse. Left: graph partitioned between two compute nodes. The gradient of the vertices indicates their LCC score. Dashed edges indicate RMA communications needed to read adjacency lists from remote nodes. Right: data reuse in social network graph [20].

Vertices with darker backgrounds have higher LCC. Given a vertex, two of its neighbors contribute to its LCC score if the three vertices form a triangle in the graph. Therefore, to compute the LCC, one has to count the number of triangles that are closed by its neighbors for every vertex.

With the rapid growth in the size of graphs to be analyzed, both memory and computational capacities of a single machine become insufficient to perform triangle counting (TC) analysis on a single node. There are two main strategies to compute TC in distributed memory: (1) by first computing overlapping partitions that are necessary for local triangle counting or (2) by issuing communications between processes. Current state-of-the-art solutions [14]–[17] all follow either the Bulk Synchronous Parallel model [18] or MapReduce [19] and suffer from synchronization, as well as the overhead of computing the partitioning of the graph. The 2020 graph champion paper TriC [16] utilizes blocking all-to-all communication resulting in synchronization overheads being as costly as communication. To minimize communication, DistTC [17] computes and distributes shadow edges that are necessary for computing triangles locally. This approach leads to a low computation time but makes the total running time dominated by this pre-computation step, similarly limiting scalability.

However, distributed triangle counting and local clustering coefficient do not necessarily require synchronization. In fact, as the graph is not updated during the computation, the distributed algorithm can be organized to let different processes progress asynchronously while still storing only partitions of the graph. We exploit this characteristic by proposing a

fully asynchronous solution that uses Remote Memory Access (RMA) one-sided operations to read remote portions of the graph without involving target nodes. For example, the graph of Figure 1 (left) is distributed on two computing nodes. To compute the LCC of vertex 1, node A reads the adjacency lists of vertices 0 and 2 locally and the adjacency list of vertex 4 via RMA.

We notice how some RMA accesses are repeated and, in principle, can be avoided to save communication time. For example, when node A needs to compute the LCC of vertex 2, it will again read via RMA the adjacency list of vertex 4. Figure 1 (right) shows the data reuse in a real-world graph modeling social network circles [20]. The plot shows how many remote reads (x -axis) are repeated y times when the graph is partitioned among two computing nodes. We exploit data reuse in the remote access pattern of LCC computations by caching RMA accesses using CLaMPI [21], a transparent caching layer for RMA. Moreover, we extend CLaMPI to accept application-defined scores for cached entries and show how this can improve caching efficiency.

All in all, in this work we:

- propose a distributed and fully asynchronous algorithm for both triangle counting and LCC (Section III-A).
- introduce a hybrid strategy for triangle computation based on the frontiers (Section III-C).
- show how data reuse in remote access patterns of LCC computation can be exploited by caching RMA accesses and reduce the overall communication time. We further increase caching performance by introducing application-defined scores for victim selection (Section III-B).

Our hybrid approach for the local computation of triangles can improve performance by up to 8% on a CPU. We achieve shared memory parallelism by computing the intersection in parallel using OpenMP, leading to a speedup of up to $2.7\times$ using 16 threads compared to a sequential implementation. On distributed memory, our non-cached algorithm achieves a speedup of up to $14\times$. Moreover, we can reduce the total running time using caching by up to 73% compared to a non-cached version until the graph is not over-partitioned. Finally, we show up to $100\times$ better performance compared to TriC.

II. BACKGROUND

A. Notation

We denote an unweighted graph that contains no multi-edges and loops as $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ the set of edges ($|V| = n$ and $|E| = m$). We will use v_i to denote a vertex and e_{ij} for the edge from v_i to v_j . We call the adjacency of v_i the set of vertices $adj(v_i) = \{v_j : e_{ij} \in E\}$ and denote by \mathbf{A} the adjacency matrix of G . Moreover, we define the in-degree of v_i as $deg^-(v_i) = |\{v_j : e_{ji} \in E\}|$ and the out-degree as $deg^+(v_i) = |\{v_j : e_{ij} \in E\}|$ (note, for a directed graph in and out-degree equals). We use the symbol Δ_{ijk} for the triangle that consists of the edges e_{ij} , e_{ik} and e_{jk} . We will denote by p the number of processes. For simplicity, we assume that p

TABLE I: Symbols used in the paper.

$G = (V, E)$	A graph G where V, E are sets of vertices and edges.
n	number of vertices.
$deg^+(v_i)$	out-degree of v_i .
$deg^-(v_i)$	in-degree of v_i .
$adj(v_i)$	adjacency of vertex v_i .
\mathbf{A}	adjacency matrix of G .
Δ_{ijk}	a triangle including the edges $e_{ij}, e_{jk}, e_{ik} \in E$
p	number of computing nodes

is always a power of 2 and p divides n . A summary of the notation used in the paper can be found in Table I.

B. Graph format

We consider graphs with no multi-edges and remove vertices that have degree less than two, as they cannot be part of any triangle. If the input graph is stored in a degree-ordered format, we use a random relabeling to avoid assigning all the highest degree vertices to the same process.

The graph is stored in the CSR (Compressed Sparse Row) format (see Figure 2), where each process stores its partition with the help of two arrays: `offsets` and `adjacencies`. An element i of the `offsets` array stores the offset at which adjacency list of vertex v_i starts in the `adjacencies` array.

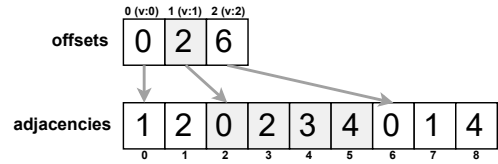


Fig. 2: Compressed Sparse Row (CSR) representation of the subgraph stored on node A of Figure 1.

C. Triangle computation

We follow the edge-centric method for triangle counting and compute the number of triangles that are closed by an edge e_{ij} for every $e_{ij} \in E$. This is given by the number of common neighbours, which can be formulated as $|adj(v_i) \cap adj(v_j)|$. We note that in the undirected case, the presence of a triangle Δ_{ijk} implies the presence of Δ_{ikj} . However, in the general edge-centric method, both triangles are enumerated because they lie on different edges. To reduce computation, this double counting can be eliminated by offsetting the neighbor's adjacency to count the common elements only for the upper triangle in the adjacency matrix, that is, for the set of vertices $\{v_k : v_k \in adj(v_j) \wedge j < k\}$.

For the computation of the intersection, we apply either binary search or sorted set intersection (SSI), which proceed for two sorted lists A and B with $|A| \leq |B|$ as follows:

1) *Binary search*: With binary search, the computation of the intersection breaks down to issuing $|A|$ lookups in a sorted array of length $|B|$ resulting in a running time complexity of $\mathcal{O}(|A| \cdot \log(|B|))$. To minimize this, one should always assign the longer list as the search tree and the shorter one as the array of keys.

Algorithm 1 Binary Search for $A \cap B$

```
1: function BINARYSEARCH( $A, B$ )
2:    $counter, bottom \leftarrow 0$ 
3:    $top \leftarrow |B| - 1$ 
4:   for all  $x \in A$  do
5:     while  $bottom < top - 1$  do
6:        $mid = \lfloor (top + bottom) / 2 \rfloor$ 
7:       if  $x < B[mid]$  then
8:          $top \leftarrow mid$ 
9:       else if  $x > B[mid]$  then
10:         $bottom \leftarrow mid$ 
11:      else
12:         $counter \leftarrow counter + 1$ 
13:      break
return  $counter$ 
```

Algorithm 2 Sorted Set Intersection for $A \cap B$

```
1: function SSI( $A, B$ )
2:    $counter \leftarrow 0$ 
3:    $i, j \leftarrow 0$ 
4:   while  $i < |A|$  and  $j < |B|$  do
5:     if  $A[i] == B[j]$  then
6:        $counter \leftarrow counter + 1$ 
7:        $i \leftarrow i + 1$ 
8:        $j \leftarrow j + 1$ 
9:     else if  $A[i] < B[j]$  then
10:       $i \leftarrow i + 1$ 
11:    else
12:       $j \leftarrow j + 1$ 
return  $counter$ 
```

2) *Sorted Set Intersection*: SSI traverses the two lists simultaneously by comparing the current elements and progressing the array whose current element is smaller. If a common element is found, it increments the counter and moves an element in both arrays. Trivially, SSI computes the intersection of two lists in $\mathcal{O}(|A| + |B|)$.

The described algorithms are outlined in Algorithm 1 and 2. We emphasize that these methods assume sorted adjacency lists, however, most graph datasets are already of this form.

D. Local Clustering Coefficient

The Local Clustering Coefficient of a vertex v_i was defined by Watts and Strogatz [9] as the proportion of existing edges between the vertices adjacent to v_i divided by the possible number of edges that can exist between them. For directed graphs, this can be computed as:

$$C(i) = \frac{|\{e_{jk} : v_j, v_k \in adj(v_i), e_{jk} \in E\}|}{deg^+(v_i) \cdot (deg^+(v_i) - 1)} \quad (1)$$

Similarly, for an undirected graph:

$$C(i) = \frac{2 \cdot |\{e_{jk} : v_j, v_k \in adj(v_i), e_{jk} \in E\}|}{deg^+(v_i) \cdot (deg^+(v_i) - 1)} \quad (2)$$

For a pair of vertices $\{v_j, v_k\}$, in order to contribute to the numerator, the edges e_{ij} , e_{ik} and e_{jk} must exist, forming the triangle Δ_{ijk} in G . Thus, if vertex degrees are known, LCC can be computed by detecting, for every vertex, the number of triangles in which they participate.

E. MPI-RMA

Remote Memory Access (RMA) operations are defined by the MPI-3 standard [22] and enable MPI processes to access memory regions of remote peers in a one-sided fashion. When running on networks supporting remote direct memory access (RDMA), RMA operations are naturally mapped to the hardware interface (e.g., ibverbs [23], uGNI [24], [25]), resulting in higher throughput and lower latency. While also two-sided communications can benefit from a RDMA-based implementation, they still incur in overheads caused by MPI message matching that can lead to additional message copies or synchronization. For this reason, in this work we focus on MPI RMA. Processes can expose their local memory by creating a window that serves as a logically distributed memory region. To access remote memories, MPI processes can use functions like `MPI_Put` and `MPI_Get`. With the first, a process can write into memory regions exposed over the network by remote peers. With the second, a process can read the content of such regions. Communications in MPI RMA are always non-blocking, and synchronization is enforced only at the beginning and at the end of an epoch. The process-local memory region can be accessed by other processes during an exposure epoch, whereas a process can access remote data during an access epoch. MPI defines two types of synchronization modes: active and passive. With the first, both initiator (i.e., the process issuing RMA operations) and target (i.e., the process targeted by RMA operations) processes synchronize to start a new epoch. With the passive mode, the participation of the target process is not required.

F. RMA Caching

CLaMPI [21] is a software caching layer that transparently caches data retrieved through MPI RMA operations. CLaMPI can be linked to MPI applications, and it is designed to fit into the MPI-3 RMA programming model, minimizing cache-management overheads and relying on the concept of MPI epochs to enforce consistency.

As applications can issue arbitrary-size read operations, CLaMPI supports caching of variable-size entries. This is achieved by using two data structures: a hash table to index cached entries and an AVL tree to store free regions in the memory buffer reserved for caching. Both the size of the hash table and the capacity of the memory buffer are parameters that can be tuned to the specific use case. CLaMPI includes an adaptive parameter tuning heuristic that automatically resizes the hash table and the memory buffer by observing indicators such as cache misses, conflicts in the hash table, and evictions due to lack of space in the memory buffer.

In CLaMPI, the eviction procedure is triggered when the application makes remote memory accesses over an MPI window that: (a) is enabled for caching, (b) does not contain the referenced data in cache, and (c) does not have enough space either in the hash table or in the memory buffer to index or store the new data. Due to the variable size of the cached entries, the system can incur in external fragmentation of the memory buffer: the free space can be fragmented in small

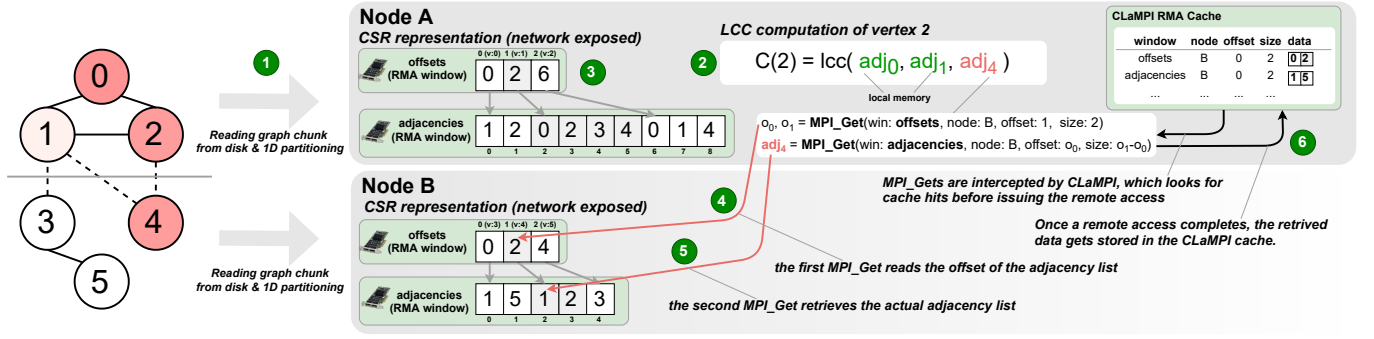


Fig. 3: Overview of the proposed approach for distributed, fully-asynchronous LCC computation with RMA caching.

non-contiguous regions that cannot fit a new entry. To reduce external fragmentation, CLaMPI assigns a score to the cached entries that reflect both their temporal locality and how much fragmentation they are causing in the memory buffer. The victim selection takes this score into account when deciding which entry to evict: e.g., if an entry is poorly placed in the memory buffer (e.g., surrounded by free space that could be merged if the entry would be removed), it will be more likely to be evicted even if it presents higher temporal locality.

CLaMPI provides three operational modes to enforce consistency of cached data: *transparent*, *always-cache*, and *user-defined*. The *transparent* mode does not make any assumption on the nature of the cached data (e.g., read-only or read-write) and flushes the cache at every epoch closure. In this case, CLaMPI can still save remote accesses that target the same data and that are made within the same epoch. However, cached data does not persist among multiple epochs. Consistency within the epoch is guaranteed by the MPI RMA semantic that, e.g., forbids conflicting *put* and *get* operations happening in the same epoch. The *always-cache* mode assumes that data accesses with RMA operations is read-only. In this case, there is no need to flush the cache since there are no updates to be propagated. Finally, the *user-defined* mode leaves the responsibility of flushing the cache to the application. For example, an application might be using data as read-only for a number of epochs, during which cached data can persist, and then switch to another phase where updates are issued, hence needing to flush the cache.

III. ACCELERATING DISTRIBUTED LCC

LCC can be formulated in terms of counting the number of closed triplets centered in a node. This formulation enables the use of triangle counting as a fundamental primitive for LCC computation. Moreover, it implies that it is possible to compute LCC of each vertex asynchronously. The problem of computing triangles in large-scale graphs has been widely investigated, and, as we mentioned, the main limitation to scalability comes from the synchronization cost and the unbalancing of the graph partitioning.

In our algorithm design, we explore asynchronous computation and a mechanism for improving data locality based on the concept of vertex delegation.

Algorithm 3 Distributed LCC Computation

- 1: **procedure** DISTRIBUTED LCC
- 2: Exchange vertices based on 1D partitioning
- 3: Build CSR representation
- 4: **for all** locally owned vertex v_i **do**
- 5: $t \leftarrow 0$ ▷ Stores the number of triangles.
- 6: **for all** e_{ij} such that $v_j \in adj(v_i)$ **do**
- 7: **if** v_j is remote **then**
- 8: RemoteRead($adj(v_j)$) ▷ See Sec. III-B
- 9: $t \ +=$ Intersect($adj(v_i), adj(v_j)$) ▷ See Sec. III-C
- 10: Compute LCC score of v_i with t triangles.

To this aim, we distribute the input graph among multiple computing nodes and let them access remote partitions via one-sided RMA operations. By distributing the graph, we lower the per-node memory requirements and reduce initialization overheads (i.e., I/O time to read the graph), which would otherwise limit strong-scaling capabilities because of Amdahl's law. Additionally, we show how the remote memory access pattern of LCC presents data reuse (temporal locality), which we exploit by caching RMA *get* operations with CLaMPI. Figure 3 shows an overview about the proposed approach.

A. Asynchronous computation

We use a 1D partitioning scheme to distribute the graph among the processes 1. In this scheme, an equal number of vertices are assigned to each process. With p computing nodes this is given by $V = V_1 \cup V_2 \dots \cup V_p$, such that:

$$V_k = \left\{ v_i : i \in \left(\frac{(k-1) \cdot n}{p}, \frac{k \cdot n}{p} \right] \right\}$$

We note that, in case the degrees are highly skewed, this partitioning method can introduce load imbalance between processes. A more balanced partitioning can be achieved by cyclic distribution [26]. However, this would require sorting vertices, introducing additional computation as well as communication.

Processes compute the number of triangles for every locally owned vertex 2. In the edge-centric method (see Section II-C), a process p_i computes $|adj(v_i) \cap adj(v_j)|$ for every $e_{i,j} \in E$ such that $v_i \in V_i$ and $v_j \in adj(v_i)$. In case the vertices v_i and v_j belong to different partitions, the owner

process first reads the adjacency of v_j from remote memory. In the CSR representation, the degree of a vertex is implicitly stored in the `offsets` array, therefore, after computing the number of triangles the LCC score is instantly attainable. The algorithm is outlined in Algorithm 3.

As the graph is stored in CSR representation ③, to read the adjacency list of a vertex from a remote node, we need to perform two remote read operations: one for reading from the `offsets` array the offset of the adjacency list in the `adjacencies` array ④, followed by a second one that reads the actual adjacency list from the `adjacencies` array (i.e., starting at the right offset) ⑤.

To enable remote access, processes expose their local graph partitions over the network. As a result, the graph is logically shared among all the processes: it can be accessed either locally (i.e., for locally owned partitions) or remotely (i.e., for partitions owned by other processes). Using MPI-RMA, this can be achieved in two windows, denoted by w_{offsets} and w_{adj} ③, in which processes share their local `offsets` and `adjacencies` arrays, respectively. When performing remote reads, processes first issue a RMA `get` targeting the w_{offsets} RMA, then issue a RMA `get` on the w_{adj} window to retrieve the adjacency list. To guarantee that the algorithm is fully asynchronous, we adopt the MPI passive target synchronization mode [22]. In this way, processes initially expose the interested memory regions in RMA windows, which then become accessible from remote peers without further synchronization. A process that wants to access a remote window first calls a `MPI_Win_lock_all` to start an access epoch. This is followed by one or more RMA `get` operations. *We remark on the unfortunate name of `MPI_Win_lock_all`: it is not an actual lock, and thus it does not synchronize processes. Instead, its effect is to signal the beginning of a new access epoch; the remote window can still be accessed by multiple (“all”) processes.* After accessing data, the process can then perform a `MPI_Win_flush` operation or close to access epoch with an `MPI_Win_unlock_all` to make sure that the remote reads are completed and that the relative data can be accessed without risk of corruption. Even in this case, the closure of an access epoch in the passive synchronization mode is a local operation, and it does not require synchronization.

Finally, to increase efficiency, we use a double-buffering approach where we overlap the processing of two consecutive edges by overlapping the computation phase of the current edge with the communication corresponding to the next one.

B. Exploiting data reuse

While remote memory accesses enable a global memory abstraction where each node can access the memory of remote ones, these accesses are normally one or more orders of magnitude more expensive than accessing local memory. For example, they can take up to 2-3 microseconds on a Cray Aries network [21]. In contrast, a DRAM accesses takes hundreds of nanoseconds that become tens of nanoseconds if the data is in cache.

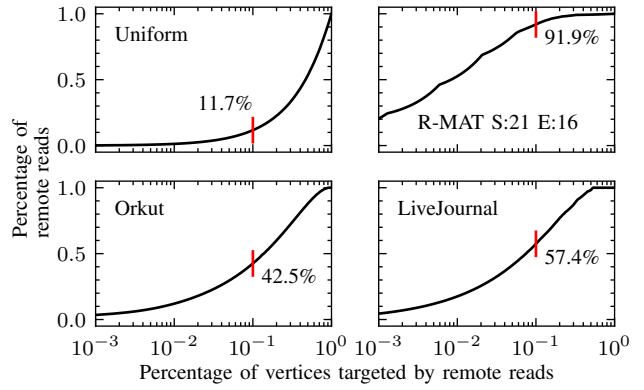


Fig. 4: Data reuse in four datasets using 8 processes and 1D partitioning. The plot shows how the highest degree vertices contribute to the total number of remote reads issued. Upper left, we show a graph with uniform degree distribution, while the other graphs follow a power law-like degree distribution (see Table II). We highlight the fraction of remote reads that target the top 10% of the highest degree vertices.

By distributing the graph among multiple processes, a large portion of the edges has endpoints in distinct partitions, thus requiring remote communications. For example, in an R-MAT graph with 2^{20} vertices and 2^{24} edges equally partitioned among 8 processes using 1D partitioning, 95% of the edges go between different partitions.

Assuming that vertices are randomly assigned to computing nodes, the in-degree of a vertex directly correlates with the number of times it will be remotely accessed. Using p computing nodes, a node will access a non-local vertex v_j in expectation $\frac{\text{deg}^-(v_j) - p}{p}$ times. Most real-world graphs present a degree distribution that follows a power law. Hence, a large portion of the remote reads will target the same small set of vertices. An illustration of how the highest degree vertices contribute to the number of remote communications can be found in Figure 4.

We exploit temporal locality in the remote memory access pattern by using a caching layer for RMA, CLaMPI, that allows to transparently cache MPI RMA accesses ⑥. By caching remote data, each node stores a dynamically defined sub-graph containing vertices that are frequently accessed and thus are expected to be accessed in the future as well.

As discussed in Section III-A, an access of a remote adjacency list is done through two steps: a first RMA `get` to read from the w_{offsets} of a remote node, followed by a second RMA `get` to the w_{adj} of the same node (using the data offset read with the first `get`). We enable caching for both RMA windows at every process. This results in two CLaMPI caches: C_{offsets} and C_{adj} . The former caches data offsets telling the position at which the adjacency list of a vertex starts and ends, whereas the latter stores full adjacency lists of cached vertices. Both caches are set to *always-cache* as the graph is never modified during the computation. In this configuration, CLaMPI does not automatically flush caches between access epochs.

1) *Cached windows characterization*: The following two observations describe the characteristics of the two CLaMPI caches and serve as a ground for their analysis:

Observation 3.1: As noted earlier, the number of accesses to a vertex correlates with its degree. As the entries in C_{adj} are adjacency lists and the size of the adjacency list of a vertex is equal to its degree, the size of the cached adjacency lists is a good indicator for the reuse of entries in C_{adj} .

Observation 3.2: On the other hand, C_{offsets} stores fixed-size entries, namely the offsets of adjacency lists in the remote `adjacencies` arrays. Therefore, there is no connection between the size of an entry and its reuse. However, an entry that stores the position of the adjacency list of a high-degree vertex will be accessed more often than an entry corresponding to a low-degree vertex.

We illustrate these observations in Figure 5 using the Facebook circles dataset [20]. The figure shows how, in the case of C_{adj} , the entry reuse (left) correlates with the entry size (right). Even though for directed graphs in- and out-degree may differ, we expect that the second observation also holds in the directed case due to the high reciprocity in real-world graphs.

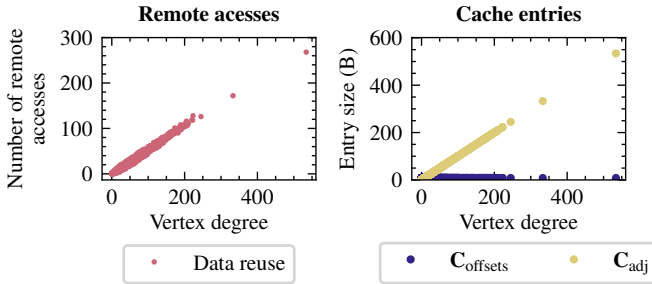


Fig. 5: Data reuse and cache entry sizes for the Facebook circles dataset [20] distributed among two computing nodes.

We enable CLaMPI’s adaptive strategy for the auto-tuning of the hash table size. However, as the adaptive strategy flushes the cache every time it adjusts the hash table size, it is crucial to determine good starting values. The size of the hash table should be equal to the expected number of entries in the cache. C_{offsets} stores fixed-size entries, where every entry corresponds to one vertex in the graph. Therefore, the number of entries in C_{offsets} is linear in n and in the size of memory allocated for the cache. For example, if the cache size equals $n/2$ bytes, the optimal size of the hash table for C_{offsets} will roughly equal $n/2$. We expect to cache some of the highest degree vertices. Thus a small number of entries in C_{adj} is likely to take up much of the space allocated for this cache (recall the size of the entry equals the degree of the corresponding vertex). If the graph’s degree distribution follows a power law, the size of the hash table for C_{adj} will be a function of the graph size and the cache size, which will also follow a power law. In this case, if the cache size is half of the graph’s size, we expect to store $n \cdot 0.5^\alpha$ many entries in the cache. We found that $\alpha = 2$ results in a good approximation for the hash table size.

2) *Application-defined scores for cached entries*: By default, CLaMPI selects entries to evict with a least recently

used (LRU) scheme weighted on a positional score to limit external fragmentation. With this scheme, high-degree vertices can still be evicted if the CLaMPI cache fills up (e.g., due to many low-degree vertices being accessed). Additionally, as CLaMPI caches a missing entry only if it has resources to store it, evicted high-degree vertices have a lower chance of being re-cached due to their larger sizes.

We notice how the vertex degree represents a good indication of the importance of storing that vertex: the higher the degree, the more probable it is that the vertex will be accessed multiple times. In particular, Observation 3.1 enables us to use this application-specific knowledge about an entry’s value to increase further caching performance. By controlling the eviction procedure based on the degree centrality of the vertices, we expect to avoid storing a high number of low-degree vertices. Lower degree vertices would consume space in the cache while having a lower likelihood of being reused. We modified CLaMPI to accept an application-specific score passed by the user. This score is used by CLaMPI in the victim selection process whenever an entry must be evicted. After completing the *get* targeting w_s , we know the out-degree of the non-local vertex, and we can assign it as a score of the respective adjacency list in C_d . We note that this extra knowledge about an entry’s value may lead to increased performance, but we lose the spatial effect of the score that attempted to reduce fragmentation.

C. Optimization of local computation

We take advantage of parallelism on the edge level and compute the intersection of the adjacency lists in parallel with OpenMP. For the binary search-based method, we distribute work among the threads by splitting the shorter (keys) array into equal-sized chunks. On the other hand, for SSI, we split the longer array and every thread intersects the assigned chunk with the shorter list. By using parallelism for the computation of the intersection and not on a higher level (e.g. distributing edges among threads) we can achieve low imbalance between the threads. However, as a too-small parallel region would limit performance, we determine a cut-off value, for which case the intersection is computed sequentially. Moreover, to decrease the cost of entering a parallel block, we specify `OMP_WAIT_POLICY=active` that forces threads to spin even if they are inactive.

Based on the time complexities of the algorithms described in Section II-C, for two sorted lists A and B with $|A| \leq |B|$, one can arrive at the following rule for the case where SSI is theoretically faster:

$$\frac{|B|}{|A|} \leq \log_2(|B|) - 1 \quad (3)$$

We utilize this decision rule to arrive at a hybrid method for triangle counting where frontiers are empirically compared to decide which method to apply for computing the intersection.

IV. EXPERIMENTAL EVALUATION

A. Experimental setup

To evaluate our distributed LCC solution, we used R-MAT [27] synthetic graphs and real-world graphs from several databases [28]–[30]. An R-MAT graph with scale x and edge factor y includes 2^x vertices and 2^{x+y} edges. We generate R-MAT graphs with the parameters $a = 0.57$, $b = c = 0.19$ and $d = 0.05$ for controlling the degree distribution. Table II lists properties of the graphs that were used for the final experiments and shows the size of their CSR representation after one-degree removal.

Shared memory benchmarks were run on an Intel® Xeon Gold 6154 @ 3.00GHz CPU with 16 cores, and the code was compiled with Intel’s ICC 2021.1 with the `-O3` flag. The distributed version was tested on the Piz Daint cluster at the Swiss National Supercomputing Centre (CSCS). We used the XC50 computing nodes, which are powered by 12 core Intel® Xeon® E5-2690 v3 2.60GHz CPUs equipped with 64GB RAM per node and interconnected with Cray’s Aries network arranged in a dragonfly topology. The code was compiled with the ICC 19.1 compiler with `-O3` flag and using the `cray-mpich 7.7.16` MPI implementation.

We distinguish small-scale experiments with no more than 64 computing nodes that we allocate on a single electrical group and large-scale experiments with 128 or more, allocated freely over the whole cluster.

Time measurements were taken using the LibLSB library [31]. For shared memory experiments, we report the median and repeated every experiment until the 5% of the median was within the 95% CI. For distributed memory experiments, we measure two different job allocations with three executions per allocation. We report the median of the longest-running node among all runs with the corresponding 95% CI. The reported results do not include the read-in of the graph and the relative distribution phase but only the time taken for the LCC computation.

TABLE II: Graphs used in this paper.

Name (type)	V	E	CSR Size
SNAP-Orkut (U)	3 M	117.2 M	905.8 MiB
SNAP-LiveJournal (U)	4 M	34.7 M	273.8 MiB
SNAP-LiveJournal1 (D)	4.8 M	69 M	273.7 MiB
SNAP-Skitter (U)	1.7 M	11.1 M	89.5 MiB
uk-2005 (D)	39.5 M	936.4 M	3.6 GiB
wiki-en (D)	13.6 M	437.2 M	1.7 GiB
R-MAT S21 EF16 (U)	2.1 M	33.6 M	251.1 MiB
R-MAT S23 EF16 (U)	8.4 M	134.2 M	1021 MiB
R-MAT S30 EF16 (U)	1073.7 M	17179.9 M	130 GiB

B. Comparison baseline

We compare our solution to TriC [16], a state-of-the-art, distributed-memory framework for global triangle counting. TriC achieves TC in a per-vertex fashion, implicitly computing LCC scores. The main difference to our solution lies in the

query-response approach used by TriC to check for necessary remote edges that leads to synchronization between processes.

TriC’s memory demand significantly increases for scale-free graphs, often leading to out-of-memory errors. In those cases, we employed a new version of TriC (*TriC Buffered*) that allocates fixed-size buffers towards remote processes.

We build TriC using the same Intel ICC 19.1 compiler and map tasks to CPU cores, as it is an MPI-only implementation. For every execution, we specify the `-b` flag to achieve a more balanced partitioning. We set the buffer size to the largest possible value not bigger than 16 MiB. This cap was necessary due to a network protocol change for messages bigger than 16 MiB that led to higher communication overheads.

C. Shared memory experiments

Our measurements comparing the different methods for computing the intersection are summarized in Table III. The trade-offs between binary search and sorted set intersection were discussed in Hu et al. [15]. They locate the main weakness of binary search on CPUs in the random accesses in the lookup tree, which leads to a high number of cache misses. On the contrary, SSI traverses the arrays linearly making possible close to zero cache misses. However, in a graph with skewed degree distribution, most edges connect vertices with degrees that are multiple orders of magnitude different from each other. In that case, binary search is essential, as its running complexity is logarithmic in the length of the longer array. Our results show the necessity of both methods on CPU, as the hybrid version always performed better than using SSI or binary search exclusively.

A strong scaling experiment was carried out to measure the gains of computing the intersection in parallel (Figure 6). We distribute the local computation on edge level that leads to

TABLE III: Performance comparison of the different intersection methods on different graphs using 16 threads. We report the number of edges processed per microsecond.

Name	Hybrid	SSI	Binary search
R-MAT S20 EF8	0.540	0.508	0.449
R-MAT S20 EF16	0.425	0.403	0.340
R-MAT S20 EF32	0.325	0.311	0.250
LiveJournal	1.084	1.018	0.984
Orkut	0.596	0.552	0.503

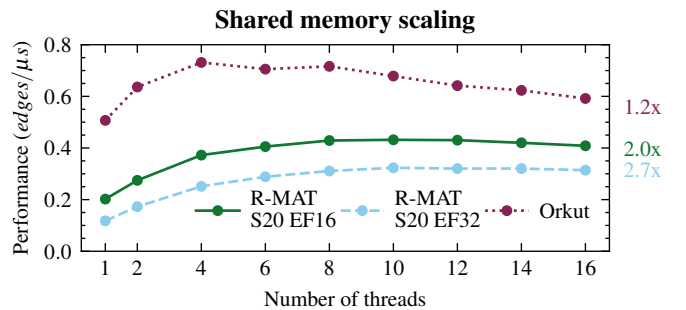


Fig. 6: Strong scaling on shared memory with hybrid method. We report the number of edges processed per microsecond.

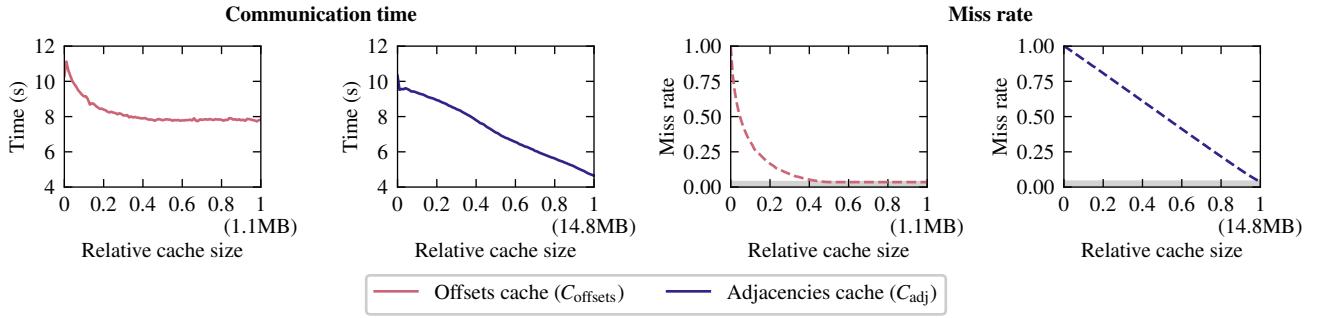


Fig. 7: Cache behaviour as a function of the cache size. We enable caching only on the respective window, and issue non-cached reads on the other. We used an R-MAT graph with 2^{20} vertices and 2^{24} edges distributed among 2 computing nodes and measured 100 configurations for both caches. The grey area shows compulsory misses.

leaving and re-entering the parallel region for every edge. In effect, the large number of OpenMP library calls becomes a performance bottleneck, which we could also justify by profiling our implementation. We could see a minor improvements of 2%-4% with using `OMP_WAIT_POLICY=active`. The best results are achieved for the R-MAT S20 EF32 graph with a speedup of $2.7\times$ from 1 to 16 threads.

D. Distributed memory experiments

1) *Caching performance*: The time of a remote read of size s bytes can be modeled by $t(s) = \alpha + s \cdot \beta$, where α is the setup overhead and β the time to read one byte. This means that for the analysis of the cache, both the number of *gets* saved by caching (hit rate) as well as the size of such *gets* have to be taken into consideration.

In Figure 7 we demonstrate how the difference between the entries stored in C_{offsets} and C_{adj} (see Section III-B) and the above remark on the duration of a remote memory read influence communication time for LCC computation. The power law-like relationship between the miss rate and C_{adj} 's size is the straight consequence of our algorithm and the graph's degree distribution. In this case, we also notice how already a small memory overhead (relative to $|V|$) allows us to save up to 30% of the time spent on communication. In contrast, we observe a linear relationship between the miss rate in C_{offsets} and its size due to the connection between an entry's size and the frequency of remote reads targeting the entry. However, a large portion of transferred data comes from remote reads targeting w_{adj} , which is reflected in the reduction in communication time achieved by C_{adj} . In this experiment, we reduce communication time by 51.6% with caching only w_{adj} . We expect that for larger datasets, this difference will grow further. Summarizing, with a small memory overhead, one can save an initial amount of communication that targets w_{offsets} . However, any further decrease will have a memory overhead linear in terms of the graph's size.

In the following, we assess the effect of the application-specific score described in Section III-B. In this experiment, we fix the memory allocated for C_{adj} to 25% of the size of the graph's non-local partition at every node to trigger the eviction procedure. The results for an R-MAT graph with 2^{20} vertices

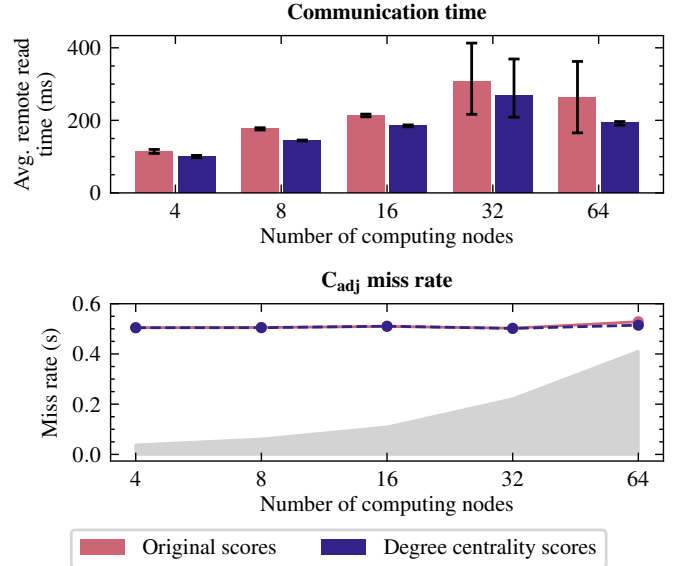


Fig. 8: Comparison of original and user-defined scores. We show the average time taken for reading a remote vertex. The grey area shows compulsory misses.

and 2^{24} edges are shown in Figure 8. Degree centrality scores improve caching performance between 14.4% and 35.6% with respect to the original scores for this dataset.

2) *Overall performance*: Finally, in Figures 9 and 10 we report the performance of our LCC implementation both with and without caching. Next to the measurements, we denote the speedup achieved with respect to the smallest configuration. For experiments using caching, a total of 16 GiB memory is reserved, allocating $0.8 \cdot |V|$ bytes for C_{offsets} and the rest for C_{adj} . Note, that with this configuration C_{offsets} can store $0.4 \cdot |V|$ many vertices, as the position of a remote adjacency list is given as a pair of $(start, end)$ positions. Furthermore, we also show measurements using TriC to better assess our implementation. In case of missing data points, the corresponding experiment exceeded a wall-time of 9 hours.

As the graph is distributed among more computing nodes, the number of edges that cause communication increases. For example, for the R-MAT S21 EF16 graph, the average fraction

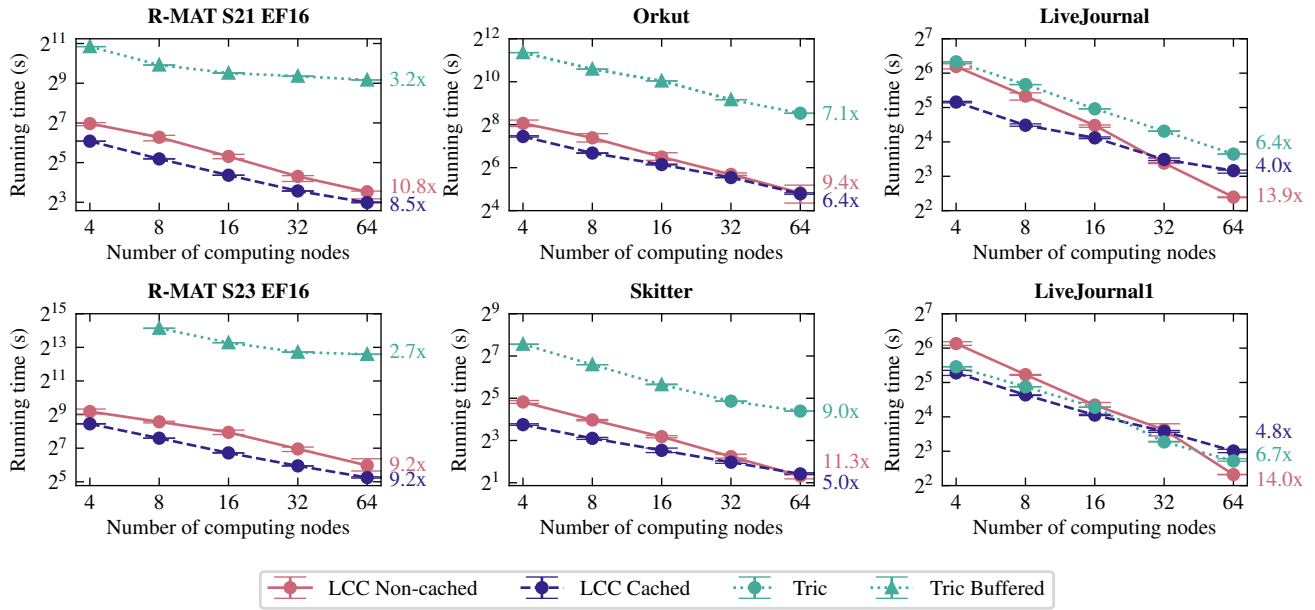


Fig. 9: Strong scaling experiments on small scale with 16 GiB memory overhead (log-log scale).

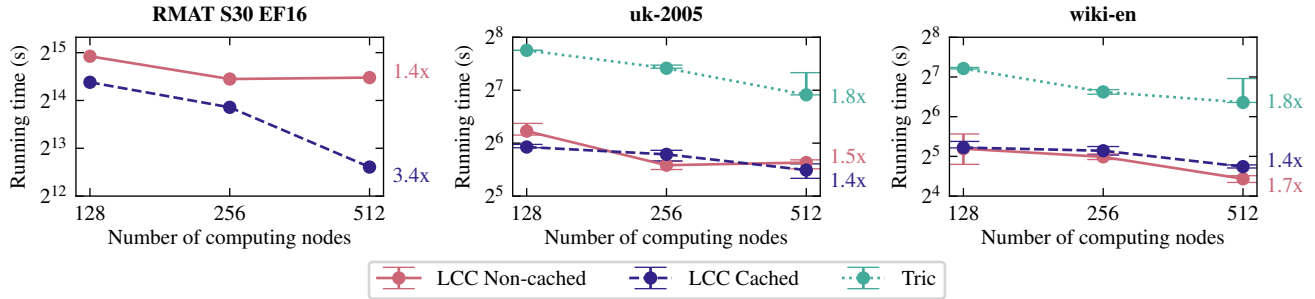


Fig. 10: Strong scaling experiments on large scale with 16 GiB memory overhead (log-log scale).¹

of remote and local reads grows from 66% to 98%. For the same graph, with 4 computing nodes, communication already takes up to 78.9% of the total running time, which rises to 97.7% for 64 nodes. In general, we could observe that communication quickly dominates the total running time, implying that the limitations of the shared memory computation have minor effects on overall performance. Moreover, it follows that overlapping communication and computation, even for more than a single edge, cannot significantly improve the runtime. As communication dominates, the speedup achievable is determined by the number of edges causing communications. At small scale, the worst scaling for a real-world graph is observed on the Orkut graph with $9.4\times$ speedup from 4 to 64 nodes. We explain this with the following two reasons. Firstly, for this graph, the running time is mostly determined by computation when 4 nodes are used, but quickly becomes communication-bound with larger configurations. Secondly, in this case, 1D partitioning causes load imbalance, leading to an up 25% difference in the running time of the different processes. We achieve our best results for the LiveJournal1 graph, with $14\times$ speedup from 4 to 64 computing nodes.

Our large-scale experiments are similarly limited by load imbalance, which bounds the achieved scaling.

The efficiency of caching is influenced by multiple factors. Firstly, in case the cache size is significantly smaller than the size of the graph, capacity misses are unavoidable. Secondly, the graph structure inherently determines the possible data reuse, and graphs with flatter degree distribution will lead to a large number of compulsory misses. Finally, by distributing the graph among an increasing number of computing nodes, data reuse reduces due to the increasing number of edges that cross partitions. This will similarly result in an increased number of compulsory misses. Compulsory misses decrease the overall hit rate and incur an overhead caused by the caching process in CLaMPI. For example, for the LiveJournal graph, 15.5% of the remote reads are compulsory misses for 4 nodes (a compulsory miss is always a compulsory miss in both caches) that grows up to 64.9% with 64 nodes.

Regarding the overall performance, we can distinguish between 3 scenarios: (1) computation dominates and caching

¹We report only one measurement per data point for the R-MAT S30 EF16 graph due to cost reasons.

has no significant effect; (2) high number of compulsory misses limit caching performance; (3) caching is beneficial and reduces the communication time. Scenario (1) can be seen for the Orkut graph where the effect of caching increases from 4 to 8 computing nodes. Scenario (2) is especially notable for the LiveJournal and LiveJournal1 graphs. In these cases, CLaMPI’s overhead leads to worse performance than the non-cached version. However, we see significant reduction in running times between these two extremes. At small scale, we achieve up to 67% and 47% better running times for the R-MAT S21 EF16 and LiveJournal graphs, respectively. At large scale, the cached version resulted in 73% better performance compared to the non-cached implementation for the R-MAT S30 EF16 graph. We remark, that this result is achieved with a cache size of only 12% of the graph’s CSR representation.

We achieve significantly better execution times both at large-scale (up to 3.6x speedup) and at small-scale (up to 100x) compared to TriC. The advantages of our asynchronous implementation over TriC is especially notable with the synthetic datasets that possess a close to perfect scale free degree distribution. Despite the imbalance between processes coming from the 1D distribution, we conclude that these results justify the necessity of an asynchronous algorithm for distributed-memory LCC computation.

V. RELATED WORK

In the following, we summarize the main techniques used for shared and distributed-memory TC analysis. For a thorough comparison of the different triangle counting algorithms, we refer the reader to the paper from Schank and Wagner [32], and to work by Shun et al. [33].

A. Frontier intersection

SSI has been introduced by Green et al. [34], and binary search appeared first for triangle counting in Hu et al. [15]. Pandey et al. [35] utilize hashing for computing intersections but, instead of hashing every element, they use a selected number of bins where multiple elements are stored. This solution was further improved in their recent work [14].

B. Algebraic computation

For a graph G one can compute the matrix $\mathbf{C} = \mathbf{A}\mathbf{A} \circ \mathbf{A}$, whose entry c_{ij} stores the number of triangles that contain e_{ij} . For undirected graphs, this can be simplified to $\mathbf{C} = \mathbf{L}\mathbf{U} \circ \mathbf{A}$, where \mathbf{L} and \mathbf{U} are the lower and upper triangular matrices. Triangle counting implementations based on this algebraic computation method take advantage of the sparsity of G and use highly optimized libraries for sparse matrix multiplication. An improved parallel implementation can be found in the paper from Azad et al. [36], and a distributed algebraic-based TC algorithm has been proposed by Hutchinson [37] using the Apache Accumulo distributed database. Aznavah et al. [38] implemented shared memory parallel TC and LCC computation based on the SuiteSparse GraphBLAS implementation of the GraphBLAS standard.

C. Distribution techniques

For any TC or LCC algorithm that utilizes parallelism on some level, work distribution is of primary importance. Kolda et al. [19] use the Mapreduce technique for triangle counting. Lumsdaine et al. [26] introduced a cyclic distribution for 1D to achieve balanced partitions. Two-dimensional partitioning assigns edges to processes in a grid-based manner. Tom and Karypis [39] developed a triangle counting algorithm for undirected graphs following a parallel matrix multiplication scheme based on 2D. Acer et al. [40] utilize 2D partitioning among the computing nodes and achieves shared memory parallelism based on 1D. Hoang et al. [17] compute shadow edges and corresponding vertices that are necessary for local triangle computation, thus avoiding any communication during the computation phase. We emphasize that all the aforementioned work requires synchronization mechanisms, and therefore, their scalability is limited.

VI. CONCLUSION

We introduce a fully asynchronous distributed-memory algorithm for both triangle counting and LCC. Synchronization overheads are removed by using RMA one-sided operations to retrieve remote parts of the graph that are needed to progress the algorithm (i.e., parts of the adjacency lists that have been partitioned and assigned to remote peers). Additionally, we show how irregular graph algorithms such as LCC and TC expose data reuse, which we exploit by using a transparent caching solution for RMA, i.e., CLaMPI. To improve cache efficiency, we extend CLaMPI to take into account application-specific scores when deciding which entries to evict in case of conflict or capacity misses. For example, by using degree centrality as the score for LCC, we are able to reduce the total running time by up to 73%. Overall, we show that removing synchronization costs and achieving vertex delegation by a caching mechanism leads to clear performance improvements over the current state-of-the-art. Finally, we plan to extend this work in many directions by i) designing new asynchronous algorithms for TC/LCC based on distribution schema that have lower communication costs than 1D distribution [41]; ii) investigating other graph problems that may benefit from the proposed approach [42]–[45] and, in general, those that can be expressed in a push-pull dichotomy [46]; iii) studying other application-specific scores for cached entries to improve caching efficiency.

ACKNOWLEDGMENT

This work has been partially funded by the UNIBZ-RTD-CALL2018-IN2087, INdAM-GNCS 2020-NoRMA, MIU-FISR-2020-FISR2020IP_00802 projects, and the European Project RED-SEA (Grant No. 955776). This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (Grant agreement No. 101002047). We thank the Swiss National Computing Center (CSCS) for providing computing resources and excellent technical support.



REFERENCES

- [1] L. Tang and H. Liu, "Graph mining applications to social network analysis," in *Managing and Mining Graph Data*. Springer, 2010.
- [2] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in bioinformatics*, vol. 7, no. 3, 2006.
- [3] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, "Complex networks: Structure and dynamics," *Physics reports*, vol. 424, no. 4-5, 2006.
- [4] D. J. Cook and L. B. Holder, *Mining graph data*. John Wiley & Sons, 2006.
- [5] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, S. Leisinger *et al.*, "Graphminesuite: Enabling high-performance and programmable graph mining algorithms with set algebra," in *VLDV*, 2021.
- [6] L. Gianinazzi, M. Besta, Y. Schaffner, and T. Hoefler, "Parallel algorithms for finding large cliques in sparse graphs," in *ACM SPAA*, 2021.
- [7] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, 2016.
- [8] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan *et al.*, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *ACM/IEEE MICRO*, 2021.
- [9] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *nature*, vol. 393, no. 6684, 1998.
- [10] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Physica A: statistical mechanics and its applications*, vol. 390, no. 6, 2011.
- [11] M. C. Nascimento, "Community detection in networks via a spectral heuristic based on the clustering coefficient," *Discrete Applied Mathematics*, vol. 176, 2014.
- [12] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rättsch, T. Hoefler, and E. Solomonik, "Communication-efficient jaccard similarity for high-performance distributed genome comparisons," in *IEEE IPDPS*. IEEE, 2020.
- [13] 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*, vol. 99, no. 9, 2002.
- [14] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li *et al.*, "TRUST: Triangle Counting Reloaded on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2021.
- [15] 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*. IEEE, 2018.
- [16] S. Ghosh and M. Halappanavar, "Tric: Distributed-memory triangle counting by exploiting the graph structure," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020.
- [17] 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)*. IEEE, 2019.
- [18] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, 1990.
- [19] T. G. Kolda, A. Pinar, T. Plantenga, C. Seshadhri, and C. Task, "Counting triangles in massive graphs with mapreduce," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, 2014.
- [20] J. J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *NIPS*, vol. 2012. Citeseer, 2012.
- [21] S. D. Girolamo, F. Vella, and T. Hoefler, "Transparent Caching for RMA Systems," in *Proceedings of the 31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)*. IEEE, May 2017.
- [22] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in mpi-3," *ACM Transactions on Parallel Computing (TOPC)*, vol. 2, no. 2, 2015.
- [23] T. Bedeir, "RDMA read and write with IB verbs," Citeseer, Tech. Rep., 2010.
- [24] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.
- [25] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with MPI-3 one sided," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.
- [26] A. Lumsdaine, L. Dalessandro, K. Dewese, J. Firoz, and S. McMillan, "Triangle counting with cyclic distributions," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020.
- [27] 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*. SIAM, 2004.
- [28] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," <http://snap.stanford.edu/data>, Jun. 2014.
- [29] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: Association for Computing Machinery, 2013.
- [30] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubcrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, 2004.
- [31] T. Hoefler and R. Belli, "Scientific Benchmarking of Parallel Computing Systems." ACM, Nov. 2015, proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).
- [32] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Experimental and Efficient Algorithms*, S. E. Nikolettseas, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.
- [33] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015.
- [34] 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*, ser. IA3 '14. IEEE Press, 2014.
- [35] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu, "H-index: Hash-indexing for parallel triangle counting on GPUs," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019.
- [36] 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*. IEEE, 2015.
- [37] D. Hutchison, "Distributed triangle counting in the graphulo matrix math library," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017.
- [38] M. Aznaveh, J. Chen, T. A. Davis, B. Hegyi, S. P. Kolodziej, T. G. Mattson, and G. Szárnyas, "Parallel GraphBLAS with OpenMP," in *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 2020.
- [39] A. S. Tom and G. Karypis, "A 2D parallel triangle counting algorithm for distributed-memory architectures," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019.
- [40] S. Acer, A. Yaşar, S. Rajamanickam, M. Wolf, and Ü. V. Catalyürek, "Scalable triangle counting on distributed-memory systems," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019.
- [41] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Euro-Par 2011 Parallel Processing*, E. Jeannot, R. Namyst, and J. Roman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
- [42] A. Formisano, R. Gentilini, and F. Vella, "Scalable energy games solvers on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, 2021.
- [43] F. Vella, M. Bernaschi, and G. Carbone, "Dynamic merging of frontiers for accelerating the evaluation of betweenness centrality," *ACM J. Exp. Algorithmics*, vol. 23, mar 2018. [Online]. Available: <https://doi.org/10.1145/3182656>
- [44] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *ACM/IEEE Supercomputing*, 2017.
- [45] M. Besta, A. Carigiet, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, and T. Hoefler, "High-performance parallel graph coloring with strong guarantees on work, depth, and quality," in *ACM/IEEE Supercomputing*, 2020.
- [46] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3078597.3078616>