Improved Distributed Algorithm for Graph Truss Decomposition

Venkatesan T. Chakaravarthy[1(✉)], Aashish Goyal[2], Prakash Murali[3], Shivmaran S. Pandian[1], and Yogish Sabharwal[1]

[1] IBM Research, Bangalore, India
{vechakra,shivmaran,ysabharwal}@in.ibm.com
[2] Indian Institute of Technology - Delhi, New Delhi, India
aashishgoyal01@gmail.com
[3] Princeton University, Princeton, USA
pmurali@cs.princeton.edu

Abstract. The truss decomposition provides a popular model for discovering cohesive communities in a given network (graph). The problem has been well studied in sequential, shared memory and MapReduce settings. We study the problem on distributed memory systems. Our work builds on two prior algorithms. The first algorithm is optimized in terms of the computational load and communication volume, but it involves a large number of iterations, leading to high load imbalance and synchronization costs. The second algorithm significantly reduces the number of iterations, but at the cost of increasing the load and the volume. We design an algorithm that offers a tradeoff between the two extremes, with the number of iterations being close to that of the second algorithm and load/volume being close to that of the first. We develop an efficient distributed (MPI) implementation based on the new algorithm. We present an experimental evaluation on large real-world graphs. The evaluation shows that the new algorithm outperforms the two prior algorithms on large system sizes with the performance gain ranging up to 2x.

1 Introduction

Discovering cohesive subgraphs or communities in a given graph is an important problem arising in diverse domains ranging from social networks to biological processes. Different models have been proposed for this purpose, among which the truss decomposition [1] is a prominent model. Apart from ensuring that the entities (vertices) in the subgraph are strongly connected among one another, the model also focuses on the strength of the connections (edges). The model is based on the intuition that the edge between two vertices can be considered strong, if they share many common neighboring entities, or alternatively, the edges are included in many triangles. For instance, in a social network, we can say that more common friends two people have, the stronger is their connection.

Given a graph G and an integer k, the k-truss is defined as the largest subgraph, in which every edge is included in at least $(k-2)$ triangles within the

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 703–717, 2018.
https://doi.org/10.1007/978-3-319-96983-1_50

subgraph. The model provides a hierarchical decomposition of the graph. The whole graph G is the 2-truss, and for $k \geq 3$, the k-truss is contained within the $(k-1)$-truss. For an edge e, the truss number $\tau(e)$ is defined as the largest k such that e belongs to the k-truss. The *truss decomposition problem* is to compute the truss numbers of all the edges.

The truss decomposition model is useful in applications such as community detection [2], visualization of large networks [3], and discovering cohesive structures containing a given set of entities [4]. The truss decomposition model builds on the prior k-core formulation [5] and the recently proposed nucleus decomposition [6] generalizes both the concepts by considering higher order cliques in place of triangles. Given the utility of the model, the truss computation is included as part of the recently proposed Graph Challenge benchmark effort (https://graphchallenge.mit.edu/).

Cohen [1] introduced the truss decomposition model and presented a polynomial time algorithm for constructing the decomposition. Building on the above work, Wang and Cheng [7] described an I/O efficient implementation. Rossi [8], Smith et al. [9], Kabir and Madduri [10,11] and Voegele et al. [12] proposed algorithms for the shared memory and GPU settings. Green et al. evaluated truss computation under GPU setting [13]. Zhang and Parthasarathy [14] independently described the model and used it as a preprocessing step for finding cliques and other dense structures. Chen et al. [15], Cohen [16] and Shao et al. [17] studied the problem on MapReduce setting.

Shared memory systems and GPUs have limitations in terms of the number of cores and/or memory availability, leading to impediments in enhancing the performance further. For instance, on a popular social network graph named `friendster` (1.8 billion edges, 4.2 billion triangles), the execution time achieved in shared memory setting (with 24 cores) is about 25 min [10]. Prior work has also considered MapReduce framework [16], but the execution times are significantly higher, due to framework overheads. Our aim is to achieve execution times of about one minute on graphs of the above size. Towards that goal, we study the problem on distributed memory systems using MPI.

We build on two prior procedures (which we adapt to the MPI setting) and study the problem from an algorithmic perspective. The first procedure, due to Cohen [1], lies at the heart of most prior implementations. While the algorithm is optimized in terms of the computational load, it takes a large number of iteration to converge. In a distributed setting the slow convergence leads to high synchronization costs and load imbalance. Working within the MapReduce framework, Chen et al. [15] proposed an algorithm which takes much lesser number of iterations, at the cost of increased computational load. We denote the two algorithms as MinTruss and PropTruss, respectively. Our main contribution is a new algorithm that offers a tradeoff between the prior algorithms in terms of the two fundamental metrics of number of iterations and load.

Truss computation is performed in two steps: a first phase that enumerates triangles and a second phase that computes the truss numbers of the edges. Triangle enumeration is a well-studied problem and efficient algorithms have been

developed (e.g., [18]). We focus on the second phase of truss computation. The second phase is iterative. In each iteration, we need to find the triangles incident on some of the edges. The prior work considers two different implementation settings. The first setting (e.g., [15,17]) explicitly stores the list of triangles enumerated in the first phase and reuses the list in the second phase, whereas the second (e.g., [8,10]) does not store the list of triangles and recomputes. The first setting has higher memory usage due to the presence of large number of triangles, but it facilitates efficient implementation of the second phase. Our implementation is based on the setting of explicitly storing the triangles. In contrast to shared memory systems, we can afford to store the list of triangles, as sufficient memory is available under the distributed memory setting.

Our Contributions

- We propose a new algorithm, denoted Hybrid, that offers a tradeoff between the prior algorithms on the two performance metrics: iterations close to PropTruss and load close to MinTruss. We present an efficient distributed memory (MPI) implementation based on the above algorithm.
- We present an experimental evaluation involving large real-world graphs (having up to 4 billion triangles). The results show that PropTruss performs the best in terms of the number of iterations. Relative to PropTruss, Hybrid is higher by at most 16x factor, whereas MinTruss is as high as 76x. In terms of load, MinTruss performs the best. Relative to MinTruss, Hybrid is higher by at most 2.3x factor, whereas PropTruss is as high as 17x.
- In terms of the execution time (truss number computation), Hybrid achieves better performance on large system sizes. On the largest system size in our study (512 MPI ranks), it outperforms MinTruss and PropTruss by up to 2x and 3.4x factors, respectively. Over the different benchmark graphs, it outperforms the best of the prior algorithms by a factor of up to 2x. The implementation is able to solve graphs having more than billion edges and 4 billion triangles in about a minute.

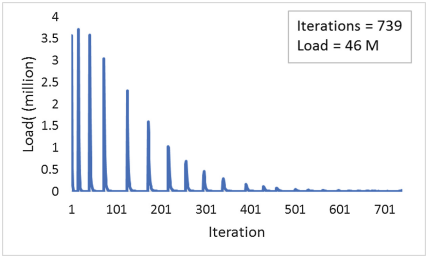
2 Preliminaries

Let $G = (V, E)$ be an undirected graph. A triple of vertices u, v and w is said to form a triangle, if $\langle u, v\rangle$, $\langle u, w\rangle$ and $\langle v, w\rangle$ are edges in G. We denote the triangle as $\Delta(u, v, w)$. The three edges are said to be *incident* on the triangle and vice versa. Two edges e and e' are called *neighbors*, if they are incident on a common triangle. Let $\gamma(G)$ denote the number triangles in G and for an edge e, let $\gamma(e)$ denote the number of triangles incident on e.

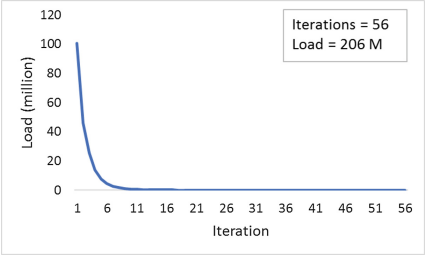
By a *subgraph*, we refer to a graph $H = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq (V' \times V') \cap E$; we denote this as $H \subseteq G$. The size of a subgraph H is measured by the number of edges in it. For a subgraph H and an edge e found in H, the *support of* e *within* H, denoted $\texttt{supp}_H(e)$, is defined as the number of triangles in H incident on e. For an integer $k \geq 2$, the *k-truss* of G is defined as the largest subgraph $H \subseteq G$ such that every edge e in H has $\texttt{supp}_H(e) \geq k - 2$ (the k-truss may not be connected). The k-truss of a graph is unique.

	Number of Iterations		Load – #updates (in millions)	
	MinT	PropT	MinT	PropT
pokec	739	56	46	206
stack	2434	134	130	1002
livej	5530	96	514	4119
orkut	4710	238	811	6447

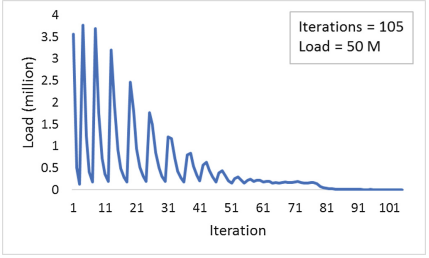
(a) Metrics



(b) MinTruss profile



(c) PropTruss profile



Hybrid profile ($\delta = 0.1$)

Fig. 1. Analysis of prior algorithms

Let κ be the largest value such that the κ-truss is non-empty. The 2-truss is simply the whole graph G. The k-trusses, for $k \geq 2$, form a hierarchical decomposition: $G = 2\text{-truss} \supseteq 3\text{-truss} \supseteq 4\text{-truss} \supseteq \cdots \supseteq \kappa\text{-truss}$. For an edge e, the *truss number of* e, denoted $\tau(e)$, is defined as the largest value k such that e is found in the k-truss. Given a graph G, the truss decomposition problem is to construct the hierarchical decomposition; equivalently, the goal is to compute the truss number $\tau(e)$ for all the edges.

3 Prior Algorithms

In this section, we present an outline of the two prior algorithms MinTruss [1] and PropTruss [15]. Both the algorithms involve a preprocessing phase, where they compute the $\mathtt{supp}_G(e)$ for all the edges via enumerating triangles of the input graph G. Triangle enumeration is a well-studied problem and efficient techniques have been developed (e.g., [18]). We describe the algorithms assuming that the supports have already been computed. For the clarity of exposition, we present the algorithms at a conceptual level, deferring distributed aspects and other implementations details to Sect. 5. A brief discussion on the preprocessing procedure can also be found in the same section.

Algorithm MinTruss: For each edge e, the algorithm maintains an upperbound $\widehat{\tau}(e)$ on the true truss number $\tau(e)$; it is initialized as $\widehat{\tau}(e) = \mathtt{supp}_G(e) + 2$. The algorithm marks all edges as *not settled* and proceeds iteratively. In each iteration, among the edges not settled, select the edges with the least truss

value and declare them to be settled. We then update the truss values of their neighbors in the following manner. Let $e = \langle u, v\rangle$ be a selected edge. For each triangle $\Delta(u, v, w)$ incident on e, if both $\langle u, w\rangle$ and $\langle v, w\rangle$ are not settled already, then decrement the truss values $\widehat{\tau}(u, w)$ and $\widehat{\tau}(v, w)$ by one. Proceed in the above manner till all the edges are settled.

Intuitively, imagine that the settled edges are deleted from the graph. The deletion of an edge e destroys the triangles incident on it. When a triangle is destroyed, the other two edges lose the support of the triangle. So, we decrement their truss values, provided e is the first edge to be deleted among the three edges. We can show that for each edge e, the truss value $\widehat{\tau}(e)$ gets decremented monotonically and becomes the true truss number $\tau(e)$ before termination.

Algorithm PropTruss: In each iteration of the MinTruss algorithm, only the neighbors of the edges with the least truss value get updated. As a result, the algorithm incurs a large number of iterations and converges slowly. Chen et al. [15] proposed an algorithm that exhibits better parallelism by taking much lesser number of iterations. We denote the algorithm as PropTruss. We rephrase and present a sketch of the algorithm.

The core idea is to select every edge e whose truss value changed in the prior iteration and propagate its new truss value to its neighbors. Since edges having various truss values propagate simultaneously, the update operation becomes more intricate, as against the simple decrement operation under the MinTruss algorithm. For a triangle $\Delta(u, v, w)$, define the truss number of the triangle as $\tau(u, v, w) = \min\{\tau(u, v), \tau(u, w), \tau(v, w)\}$. The new update operation is based on the following proposition. The truss numbers can be seen as stationary solutions satisfying the condition given by the proposition.

Proposition 1. *For any edge $e = \langle u, v\rangle$, we have that*

$$\tau(e) = \max\{j \ : \ |\{\Delta(u, v, x) \ : \ \tau(u, v, x) \geq j\}| \geq j - 2\}$$

For each triangle $\Delta(u, v, w)$, the algorithm maintains an upperbound $\widehat{\tau}(u, v, w) \geq \min\{\widehat{\tau}(u, v), \widehat{\tau}(u, w), \widehat{\tau}(v, w)\}$. These are initialized to ∞. We ensure that for any edge $e = \langle u, v\rangle$, a condition analogous to the proposition is true throughout the execution of the algorithm:

$$\widehat{\tau}(e) = \max\{j \ : \ |\{\Delta(u, v, x) \ : \ \widehat{\tau}(u, v, x) \geq j\}| \geq j - 2\} \tag{1}$$

The PropTruss algorithm can be summarized as follows. In each iteration, consider all the edges $e = \langle u, v\rangle$ whose truss value changed in the prior iteration. For each triangle $\Delta(u, v, w)$ incident on e, if $\widehat{\tau}(e) < \widehat{\tau}(u, v, w)$, then we update the truss value of the triangle to $\widehat{\tau}(e)$. As a result, the truss values of the edges $\langle u, w\rangle$ and $\langle v, w\rangle$ may no longer satisfy condition (1). So, for both the edges, we recompute the right hand side and update their truss values accordingly. We proceed in this manner, until a stable solution is reached, wherein the truss value of none of the edges changes. In the first iteration, all the edges get selected and perform the above propagate operation.

Comparison of MinTruss **and** PropTruss: We compare the algorithms using two fundamental metrics: (i) number of iterations; (ii) *load* - the total number of updates (one update is counted whenever an edge changes the truss value of a triangle and propagates to the other two edges of the triangle). In a distributed setting, higher number of iterations leads to higher synchronization cost and load imbalance. The second metric determines the computational load and the communication volume.

The PropTruss algorithm is superior on the first metric, because edges from multiple truss levels propagate their truss value simultaneously leading to faster convergence. On the other hand, the MinTruss algorithm is better in terms of load. The reason is that any edge e propagates its truss value only once during the entire execution (when its truss value $\widehat{\tau}(e)$ settles to the true truss number $\tau(e)$), whereas the same edge may propagate multiple times under PropTruss.

Figure 1(a) illustrates the above tradeoff by providing the two metrics on four sample graphs drawn from our experimental evaluation (properties are graphs can be found in Sect. 6). We can see that PropTruss involves significantly lesser number of iterations, but MinTruss is superior on load.

4 Algorithm Hybrid

In this section, we present a new algorithm, denoted Hybrid, that strikes a tradeoff between the two prior algorithms. It aims at achieving load close to MinTruss and the number of iterations close to PropTruss.

The new algorithm is motivated from an analysis of prior algorithms in terms of their load profiles, a plot that shows the load incurred in each iteration of the algorithm. As an illustration, Fig. 1(b) and (c) provide the load profiles of the two algorithms on the `pokec` graph. We can see that PropTruss incurs the maximum load in the first iteration and the load monotonically decreases until the algorithm converges. On the other hand, in the case of MinTruss, the iterations are grouped into many blocks; within each block the load is maximum in the initial iteration and then decreases monotonically. Each block corresponds a truss value k and all the edges with the truss number $\tau(e) = k$ settle in the successive iterations of the block. While the MinTruss algorithm involves a large number of iterations, most of the iterations incur very little load. The core idea behind the Hybrid algorithm is to eliminate the low-load iterations, without compromising much on the overall load incurred.

Algorithm Hybrid: Like the prior algorithms, we maintain an upperbound $\widehat{\tau}(e)$ on the true truss number $\tau(e)$, for all edges e, and initialize it to $\mathtt{supp}_G(e)+2$. Let $k_{\min}$ and $k_{\max}$ denote the minimum and the maximum truss value $\widehat{\tau}(e)$ among all the edges e. We imagine that each truss value is a bucket and each edge e resides in the bucket corresponding to its truss value $\widehat{\tau}(e)$. As the algorithm proceeds, whenever $\widehat{\tau}(e)$ decreases, we visualize that the edge moves from its current bucket to a lower bucket. We maintain a set of edges called the *active set*, denoted `Act`. The edges in the set would propagate their truss values in each iteration. The edges belonging to the active set are drawn from a window of

Pre-processing: Compute $\texttt{supp}_G(e)$ for all edges.
Initialization:
For each triangle $\Delta(u,v,w)$, set $\widehat{\tau}(u,v,w) \leftarrow \infty$
For all $e = \langle u,v\rangle \in E$
  Set $\widehat{\tau}(e) \leftarrow \texttt{supp}_G(e) + 2$.
  Set $g_e \leftarrow \widehat{\tau}(e) - 2$ and for $0 \le j < \widehat{\tau}(e)$, set $h_e(j) \leftarrow 0$
Truss Computation:
$k_{\min} \leftarrow \min\{\widehat{\tau}(e) \ : \ e \in E\}$ and $k_{\max} \leftarrow \max\{\widehat{\tau}(e) \ : \ e \in E\}$
Window $W \leftarrow [k_{\min}, k_{\min}]$
$\texttt{Act} \leftarrow \{e \ : \ \widehat{\tau}(e) \in W\}$ /* Active set */
$\gamma_{\max} \leftarrow 0$
Loop /* Iterations */
  if($\texttt{Act} = \emptyset$ and $W = [k_{\min}, k_{\max}]$) then **terminate**.
  Execute procedure *Window-Expansion.*
  For each $e = \langle u,v\rangle \in \texttt{Act}$
    For each triangle $\Delta(u,v,w)$ with $\widehat{\tau}(e) < \widehat{\tau}(u,v,w)$
      $\text{val}_{\text{old}} \leftarrow \widehat{\tau}(u,v,w)$ and $\widehat{\tau}(u,v,w) \leftarrow \widehat{\tau}(e)$ and $\text{val}_{\text{new}} \leftarrow \widehat{\tau}(u,v,w)$
      Update($\langle u,w\rangle, \text{val}_{\text{old}}, \text{val}_{\text{new}}$) and Update($\langle v,w\rangle, \text{val}_{\text{old}}, \text{val}_{\text{new}}$)
  $\texttt{Act} \leftarrow \{e : \widehat{\tau}(e) \in W$ and $\widehat{\tau}(e)$ changed in the current iteration$\}$
  Let $\gamma(\texttt{Act}) = \sum_{e \in \texttt{Act}} \gamma(e)$, where $\gamma(e)$ is the number of triangles on e
  $\gamma_{\max} \leftarrow \max\{\gamma_{\max}, \gamma(\texttt{Act})\}$.
Procedure *Window-Expansion:*
Let k be the last bucket in W.
while($(\gamma(\texttt{Act}) \le \delta \cdot \gamma_{\max})$ and $(k \ne k_{\max})$)
    $W = [k_{\min}, k+1]$ and $\texttt{Act} \leftarrow \texttt{Act} \cup \{e \ : \ \widehat{\tau}(e) = k+1\}$ and $k \leftarrow k+1$
Procedure Update($e', \text{val}_{\text{old}}, \text{val}_{\text{new}}$)
  Case 1 [$\text{val}_{\text{old}} \ge \widehat{\tau}(e')$ and $\text{val}_{\text{new}} \ge \widehat{\tau}(e')$]: do nothing.
  Case 2 [$\text{val}_{\text{old}} \ge \widehat{\tau}(e')$ and $\text{val}_{\text{new}} < \widehat{\tau}(e')$]: Decr. $g_{e'}$; incr. $h_{e'}(\text{val}_{\text{new}})$
  Case 3 [$\text{val}_{\text{old}} < \widehat{\tau}(e')$ and $\text{val}_{\text{new}} < \widehat{\tau}(e')$]: Decr. $h_{e'}(\text{val}_{\text{old}})$; incr. $h_{e'}(\text{val}_{\text{new}})$
  if($g_{e'} < \widehat{\tau}(e') - 2$) then decr. $\widehat{\tau}(e')$ and $g_{e'} \leftarrow g_{e'} + h_{e'}(\widehat{\tau}(e'))$.

Fig. 2. Algorithm Hybrid

buckets, denoted W. To start with, the window consists of only the bucket $k_{\min}$, i.e., $W = [k_{\min}, k_{\min}]$. In each iteration, we construct the active set by including all edges e such that $\widehat{\tau}(e)$ changed in the prior iteration and e belongs to one of the buckets in the window.

In the next and the crucial step, we use an appropriate heuristic to estimate whether the current active set would result in the load being too low. In this case, we expand the window by including the next bucket, and add all the edges in the bucket to the active set. We repeat the above process until the heuristic determines that the load would be sufficiently high.

We proceed in the above manner until all the buckets have been added and the window becomes the complete range $[k_{\min}, k_{\max}]$. At this stage, we continue with the iterations until the active set becomes empty; namely, the truss value does not change for any of the edges. A pseudocode for Hybrid is given in Fig. 2.

Window Expansion Heuristic: We develop a heuristic for window expansion by estimating the load to be incurred on the current active set $\texttt{Act}$. Let $e = \langle u, v\rangle$ be an edge in $\texttt{Act}$. For each triangle $\Delta(u, v, w)$ incident on e, we update the two neighboring edges provided $\widehat{\tau}(u, v) < \widehat{\tau}(u, v, w)$; let $\widetilde{\gamma}(e)$ denote the number of such triangles. The exact load under $\texttt{Act}$ is the sum of $\widetilde{\gamma}(e)$ for all edges $e \in \texttt{Act}$. Unfortunately, $\widetilde{\gamma}(e)$ changes dynamically and its computation requires an expensive scan of the triangles incident on e. We avoid the scan by using the upperbound $\gamma(e)$ (the number of triangles incident on e). In contrast to $\widetilde{\gamma}(e)$, the quantity $\gamma(e)$ is static and can be computed as part of the preprocessing stage. Define $\gamma(\texttt{Act}) = \sum_{e \in \texttt{Act}} \gamma(e)$. We take $\gamma(\texttt{Act})$ as an estimate on the load incurred by the set $\texttt{Act}$.

We determine if the above estimate is high enough by comparing against the maximum number of triangles encountered in the prior iterations. Meaning, let $\texttt{Act}_j$ denote the active set in a prior iteration j and let $\gamma(\texttt{Act}_j)$ denote aggregate number of triangles incident on the edges in $\texttt{Act}_j$. We keep track of the quantity $\gamma_{\max} = \max_j \gamma(\texttt{Act}_j)$. The heuristic estimates that the load on $\texttt{Act}$ would be low, if the ratio of $\gamma(\texttt{Act})$ to $\gamma_{\max}$ is below a threshold δ. In this case, we expand the window by including the next bucket. The process is repeated until the estimate on the load becomes sufficiently high. In the above procedure, δ is a tunable parameter. Pseudocode for the procedure can be found in Fig. 2.

Update Operation: As in the case of the PropTruss algorithm, our update operation is also based on Proposition 1. Recall that in the PropTruss algorithm, whenever an edge $e = \langle u, v\rangle$ updates the truss value $\widehat{\tau}(u, v, w)$ for a triangle $\Delta(u, v, w)$, the truss values are recomputed for the other two edges $\langle u, w\rangle$ and $\langle v, w\rangle$ via evaluating the right hand side of condition (1). We develop a more efficient method that avoids the expensive recomputation by maintaining suitable histograms, as described below.

Consider any edge $e = \langle u, v\rangle$. We group the triangles incident on e based on their truss values and maintain a histogram consisting of two components, $h_e(\cdot)$ and g_e. For $j < \widehat{\tau}(e)$, $h_e(j)$ stores the number of triangles with truss value exactly j, whereas g_e keeps track of the number of triangles with the truss values at least $\widehat{\tau}(e)$. Namely:

$$\forall j < \widehat{\tau}(e): \quad h_e(j) = |\{\Delta(u, v, x) : \widehat{\tau}(u, v, x) = j\}| \tag{2}$$

$$\text{and} \quad g_e = |\{\Delta(u, v, x) : \widehat{\tau}(u, v, x) \geq \widehat{\tau}(e)\}| \tag{3}$$

For each triangle $\Delta(u, v, w)$, we initialize $\widehat{\tau}(u, v, w) = \infty$. For each edge e, the histogram is initialized as $g_e = \widehat{\tau}(e) - 2$ and for all $j < \widehat{\tau}(e)$, $h_e(j) = 0$.

The iterations are executed as follows. Consider each edge $\langle u, v\rangle$ found in the active set. For each triangle $\Delta(u, v, w)$ incident on e, if $\widehat{\tau}(e) < \widehat{\tau}(u, v, w)$, we update $\widehat{\tau}(u, v, w) = \widehat{\tau}(e)$. Let val_{old} denote the value of $\widehat{\tau}(u, v, w)$ before the update was performed and val_{new} be the new value $(= \widehat{\tau}(e))$. We update the histogram and $\widehat{\tau}(\cdot)$ value for the other two edges $\langle u, w\rangle$ and $\langle v, w\rangle$ in such a manner that the conditions (1), (2) and (3) continue to be satisfied.

Let e' be one of other two edges. Before the update, the triangle is counted as part of $g(e')$, if $\text{val}_{\text{old}} \geq \widehat{\tau}(e')$ and as part of $h_{e'}(\text{val}_{\text{old}})$, if $\text{val}_{\text{old}} < \widehat{\tau}(e')$. Similarly, after the update the triangle is counted as part of $g(e')$, if $\text{val}_{\text{new}} \geq \widehat{\tau}(e')$ and as part of $h_{e'}(\text{val}_{\text{new}})$, if $\text{val}_{\text{new}} < \widehat{\tau}(e')$. Thus, based on the value of val_{old} and val_{new}, we adjust (increment/decrement) $g(e')$, $h_{e'}(\text{val}_{\text{old}})$ and $h_{e'}(\text{val}_{\text{new}})$; see Fig. 2. We then decrement $\widehat{\tau}(e')$, if $g_{e'} < \widehat{\tau}(e') - 2$. Furthermore, in this case, $h_{e'}(\widehat{\tau}(e'))$ must now be counted as part of $g_{e'}$ and we add $h_{e'}(\widehat{\tau}(e'))$ to $g_{e'}$. Our implementation of PropTruss also uses the above histogram strategy.

Discussion: The two prior algorithms can be realized by modifying the window expansion heuristic: PropTruss via initializing the window to include all the buckets; MinTruss via expanding the window with the next bucket only when the active set becomes empty. By tuning the parameter δ, we get a spectrum of algorithms offering tradeoff between the two extremes. On one hand, restricting the active set to a window of buckets leads to lesser load than PropTruss. On the other hand, ensuring that the load is high enough in each iteration leads to faster convergence and lesser number of iterations than MinTruss. We can prove the following tradeoff for any value of $\delta \in [0, 1]$:

$$\text{Number of iterations} : \mathsf{PropTruss} \leq \mathsf{Hybrid} \leq \mathsf{MinTruss}$$
$$\text{Load} : \mathsf{MinTruss} \leq \mathsf{Hybrid} \leq \mathsf{PropTruss}$$

Figure 1(c) shows the load profile for the `pokec` graph with $\delta = 0.1$. We can see that the number of iterations is close to PropTruss and the load is close to MinTruss. The profile also exhibits a blocked behavior, but the load in any iteration is sufficiently high.

At a high level, computing the truss decomposition shares similarities with the single source shortest path problem (SSSP). Similar to truss computation, prior algorithms for SSSP maintain an upperbound on the shortest distances which get iteratively refined. Here, we can draw parallels between edges and the truss numbers on one hand, and the vertices and the shortest distances on the other. Viewed from this perspective, the MinTruss and the PropTruss algorithms are analogous to the well-known Dijkstra's and the Bellman-Ford algorithms, respectively. The Hybrid algorithm is inspired by the Δ-stepping algorithm [19].

5 Distributed Implementation

Graph Distribution: We distribute the input graph $G = (V, E)$ among the processors (MPI ranks) using a degree-based ordering proposed in prior work in the context of efficient triangle counting (e.g., [18]). For a vertex u, let $\texttt{deg}(u)$ denote its degree. Arrange the vertices in the increasing order of degrees, breaking ties via lexicographic identifiers. Namely, we say that $u \prec v$, if either $\texttt{deg}(u) < \texttt{deg}(v)$, or $\texttt{deg}(u) = \texttt{deg}(v)$ and $\texttt{id}(u) < \texttt{id}(v)$. Let $\texttt{deg}_+(u)$ be the number of neighbors of u with $v \succ u$.

We assign each vertex u to a processor chosen uniformly at random, called the owner of u. We also assign ownership for each edge $e = \langle u, v \rangle$: assign e to

the owner of u, if $u \prec v$, and to the owner of v, if $v \prec u$. Let $V(p)$ and $E(p)$ denote the set of vertices and edges owned by a processor p.

For a processor p, let $\gamma(p)$ denote the aggregate number of triangles incident on the edges owned by p, i.e., $\gamma(p) = \sum_{e \in E(p)} \gamma(e)$. The quantity $\gamma(p)$ is an indicator of the number of updates performed by the processor during the truss computation. We can derive a bound on $\gamma(p)$ follows. For each vertex $u \in V(p)$, the processor owns $\mathtt{deg}_+(u)$ edges incident on u; each of these edges can be incident on at most $\mathtt{deg}(u)$ triangles. Hence, $\gamma(p)$ is at most $\sum_{u \in V(p)} \mathtt{deg}(u)\mathtt{deg}_+(u)$. Intuitively, if u is a low-degree vertex, then $\mathtt{deg}_+(u)$ is also low, whereas if u is a high-degree vertex, then it cannot have too many neighbors succeeding it in the ordering and so, $\mathtt{deg}_+(u)$ is again low. As a result, the above distribution helps in achieving good load balance.

Preprocessing - Triangle Enumeration: All the three algorithms involve a preprocessing stage of computing the support of the edges, via triangle enumeration. For this purpose, we adopt an efficient strategy proposed in prior work (e.g., [18]). We say that a pair of edges $\langle u, v\rangle$ and $\langle u, w\rangle$ is a *monotone wedge*, $v \succ u$ and $w \succ u$. The strategy is to enumerate all the monotone wedges $\langle u, v\rangle$ and $\langle u, w\rangle$ and test whether $\langle v, w\rangle$ is also an edge. The advantage with this approach is that the number of wedges considered is only $\sum_{u \in V} \mathtt{deg}_+^2(u)$.

In our distributed implementation, each processor p builds a hash table over edges $E(p)$ owned by it. For each vertex $u \in V(p)$, the processor p enumerates all monotone wedges $\langle u, v\rangle$ and $\langle u, w\rangle$, and sends the triple (u, v, w) to the processor owning v, say q. Using its hash table, the processor q checks if the pair $\langle v, w\rangle$ is an edge in G and if so, the triangle $\Delta(u, v, w)$ has been discovered. In this case, q increments $\mathtt{supp}_G(v, w)$ and sends the triple (u, v, w) back to p, upon receiving which p increments both $\mathtt{supp}_G(u, v)$ and $\mathtt{supp}_G(u, w)$. In the above process, for each edge e, its owner stores the list of triangles incident on e.

Truss Computation: The algorithms are implemented under the bulk synchronous parallel model. For each edge $e = \langle u, v\rangle$, the owner of e maintains $\widehat{\tau}(e)$, histogram $h_e(\cdot)$ and g_e. In addition, for each triangle $\Delta(u, v, w)$ incident on e, the processor also stores a local copy of $\widehat{\tau}(u, v, w)$. In each iteration, for each edge $e = \langle u, v\rangle \in \mathtt{Act}$, the owner of e propagates the new truss value $\widehat{\tau}(e)$, as follows. For each triangle $\Delta(u, v, w)$ with $\widehat{\tau}(e) < \widehat{\tau}(u, v, w)$, p sends update messages to the owners of the edges $\langle u, w\rangle$ and $\langle v, w\rangle$, wherein the message consists of the identification of the triangle $\Delta(u, v, w)$, as well as the new value of $\widehat{\tau}(u, v, w)$. The messages are exchanged using the $MPI\_Alltoallv$ primitive. Each processor executes the update procedure on the received messages, updating the edge truss values, histograms, as well as the local copies of the triangle truss values. The buckets and the active sets are stored in a distributed manner: each processor p maintains the buckets and active sets restricted to the edges owned by it.

6 Experimental Evaluation

Experimental Setup: The experiments were conducted on a cluster of Power-8 nodes (20 physical cores, 512 GB memory, 4 GHz) connected via InfiniBand in

Graphs	n	m	Δ	κ
pokec	1.6	31	34	29
stackoverflow	2.6	36	114	79
livejournal	4.8	69	292	362
orkut	3.1	117	633	78
flickr	2.3	33	841	297

Graphs	n	m	Δ	κ
gplus	0.11	14	1076	418
uk-2002	18.5	298	4607	944
hollywood -2009	1.14	114	4918	2209
friendster	65.6	1806	4244	129

Fig. 3. Graph properties: number of vertices (n), edges (m) and triangles (Δ), all in millions. The maximum truss number κ is also shown.

	Iterations			Normalized load			Normalized max-load		
	MinT	Hybrid	PropT	MinT	Hybrid	PropT	MinT	Hybrid	PropT
stack	2434	471	134	1.1	1.4	8.8	13.3	7.5	16.9
livej	5530	600	96	1.8	4.1	14.1	83.9	37.7	25.9
orkut	4710	523	238	1.3	1.5	10.2	5.0	3.1	12.2
flickr	10188	2739	178	1.2	2.5	19.2	77.5	45.9	42.8
gplus	13574	3597	222	1.2	2.7	21.5	114.3	75.9	90.3
uk	35080	4407	4404	2.0	2.4	6.5	13.8	6.9	8.8
hollywood	12629	1657	165	2.6	2.8	7.0	30.0	17.2	14.6
friendster	7430	706	706	1.2	1.5	7.4	1.7	1.6	7.6

Fig. 4. Basic metrics

a fat-tree topology. We launch 16 MPI ranks per node, each mapped to a core. We use 2 to 32 nodes, leading to a total of 32 to 512 MPI ranks.

The dataset consists of eight representative real-world graphs obtained from the SNAP repository[1], the Koblenz network collection[2] and the University of SuiteSparse Matrix Collection[3]; the `uk-2002` and `hollywood-2009` graphs are based on the prior work [20]. Four of the graphs are medium-sized with more than 100 million triangles, and the other four are large graphs with more than billion triangles. Figure 3 shows the properties of the graphs, including the small `pokec` graph used as a case study in earlier discussion (the graphs are sorted by the number of triangles).

Prior work has presented efficient shared memory implementations for truss computation [10,13]. These are based on the MinTruss algorithm and provide optimizations for the above setting. Our objective is to study the two extremes of MinTruss and PropTruss, and the effect of the tradeoff offered by Hybrid under distributed memory setting. Towards the objective, our experimental evaluation focuses on the three algorithms.

Recall that Hybrid offers a tradeoff between MinTruss and PropTruss, controlled by δ. We experimented with different values of the parameter on different graphs and system sizes, and found that setting $\delta = 0.1$ offers the best tradeoff. All the experiments discussed below use the above setting of the parameter.

Basic Metrics: We first evaluate the algorithms on the two basic metrics: number of iterations and load (number of updates). We normalize the load by $\gamma(G)$, the number of triangles in the graph. An ideal value for normalized load

[1] http://snap.stanford.edu/data.
[2] http://konect.uni-koblenz.de/.
[3] https://sparse.tamu.edu/.

is one unit, which is attained when an algorithm performs only a single update per triangle.

The results, shown in Fig. 4, confirm our earlier analysis (Sect. 3). We can see that MinTruss incurs a large number of iterations, whereas PropTruss takes much lesser number of iterations, with the reduction being as high as 76x (on `hollywood`). The above trend is reversed on the metric of load. The MinTruss algorithm performs the best with near-ideal load, whereas the quantity is as high as 22 units for PropTruss. The Hybrid algorithm strikes a balance between the two algorithms. In terms of the number of iterations, relative to PropTruss, Hybrid is higher by at most 16x factor (whereas MinTruss is as high as 76x). In terms of load, relative to MinTruss, Hybrid is higher by at most 2.3x factor (whereas PropTruss is as high as 17x).

Another metric of importance is the max-load, which quantifies the load balance characteristics. We compute the max-load by finding the maximum load among the processors in each iteration and summing up across all the iterations. An ideal value of the metric is $\gamma(G)/P$, where P is the number of processors; We normalize the max-load by this quantity. Figure 4 presents the normalized max-load at $P = 512$ (the largest system size in our study). In spite of achieving near-ideal load, the MinTruss algorithm incurs the highest max-load in most cases. The reason is that the load gets spread over the large number of iterations, leading to load imbalance. The PropTruss and the Hybrid algorithms involve lesser number of iterations and perform comparatively better.

stack	PreP	MinT	Hybrid	PropT
32	10.5	4.7	5.7	28.3
64	5.0	2.6	2.8	12.6
128	2.4	1.7	1.5	6.2
256	1.1	1.3	1.0	3.2
512	0.6	1.4	0.7	2.0

livej	PreP	MinT	Hybrid	PropT
32	18.3	22.1	30.9	94.5
64	8.9	15.5	16.7	43.3
128	4.2	12.5	10.4	20.9
256	1.9	10.9	7.0	10.6
512	0.9	10.8	5.3	5.9

orkut	PreP	MinT	Hybrid	PropT
32	64.5	30.9	39.5	235
64	31.2	14.7	17.9	105
128	15.2	7.5	8.4	46.9
256	7.5	4.8	4.3	21.4
512	3.4	3.4	2.4	10.3

flickr	PreP	MinT	Hybrid	PropT
32	55.4	69.7	96.9	562.6
64	26.5	49.7	54.93	256
128	12.8	39.4	35.11	123
256	6.29	35.1	25.3	60.6
512	3.3	33.2	20.8	34.7

gplus	PreP	MinT	Hybrid	PropT
32	77.8	119	164	990
64	44.3	93.4	110	575.0
128	25.5	78.1	78.6	350.3
256	16	70.5	64.8	226.3
512	8.74	65.6	50.0	118.2

uk	PreP	MinT	Hybrid	PropT
32	344.2	266	309	599
64	165	151	161.2	341
128	78.2	91.2	85.1	175
256	38	61.3	48.9	93.2
512	17.5	46.8	28.4	44.9

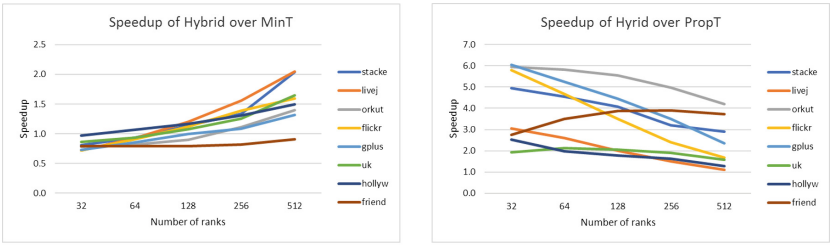
hollyw	PreP	MinT	Hybrid	PropT
32	335	416	428	1081
64	167	272.3	255.5	507
128	85.8	179	153.8	272
256	48.6	132.8	101.5	166
512	24.3	100.9	67.5	86.5

friend	PreP	MinT	Hybrid	PropT
32	1219	359	453	1246
64	584	148	187	653
128	266	70.2	88.6	343
256	130	33.9	41.5	162
512	62.7	17.5	19.2	71.5

Fig. 5. Execution time (seconds) on the benchmark graphs on ranks from 32 to 512. The best running times are highlighted.

Truss Computation: Execution Time: We next evaluate the truss computation time of the algorithms on different systems sizes (32 to 512 ranks). The results are shown in Fig. 5 (the running times are for a single run of the algorithms). The best execution time is highlighted for each configuration. The figure also includes the preprocessing time (triangle enumeration), which is common for all the algorithms.

We can observe that the MinTruss algorithm performs the best on small system sizes. However, as the system size increases, the algorithm suffers from



Average	32	64	128	256	512
MinT	0.8	0.9	1.0	1.2	1.6
PropT	4.1	3.8	3.4	2.9	2.4

Fig. 6. Speedup of Hybrid over MinTruss and PropTruss. The average speedup on the eight graphs at different ranks are also shown.

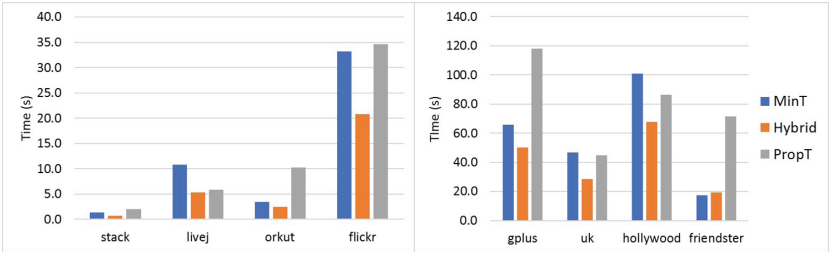


Fig. 7. Truss computation time(s) at 512 ranks

synchronization costs and load imbalance arising out of the large number of iterations, resulting in degradation of the performance. Except `friendster`, the Hybrid algorithm outperforms both the prior algorithms on larger systems sizes.

The `friendster` graph is one of the largest in terms of the number of triangles. However, the maximum truss number κ is comparatively smaller leading to lesser number of iterations for MinTruss. Consequently, the synchronization cost and load imbalance are lesser, and so, MinTruss outperforms Hybrid on all the system sizes in the study. We expect Hybrid to outperform MinTruss at system sizes larger than 512 ranks.

Figure 6 provides the speedup of Hybrid over MinTruss and PropTruss on the different graphs, as the number of ranks is varied from 32 to 512. The speedup is measured as a ratio of the running time of the competing algorithm (MinTruss or PropTruss) to that of Hybrid. The figure also provides the average speedup over the eight benchmark graphs across 32 to 512 ranks. With respect to MinTruss, the speedup is less than one on small systems sizes (since MinTruss is superior). On the largest system size of 512, Hybrid outperforms MinTruss, with the speedup ranging up to 2x with the average being 1.6x. With respect to PropTruss, Hybrid achieves better speedup at smaller ranks. As the number of ranks increases, the speedup decreases because of increase in synchronization cost and load imbalance

under Hybrid. Nevertheless, we see that on the largest system size of 512, the speedup is up to 4.2x with the average being 2.4x.

Figure 7 compares the execution times on the largest system size of 512 ranks. We can see that Hybrid outperforms MinTruss and PropTruss by factors of up to 2x (on `stackoverflow`) and 4x (on `orkut`), respectively. Taking the best of the prior algorithms in each case, the performance gain is up to a factor of 2x (on `stackoverflow`).

7 Conclusions

We presented a new distributed algorithm for truss decomposition that offers a tradeoff between two prior procedures in terms of the metrics of number of iterations and the number updates. Our experimental study shows that the algorithm outperforms the prior procedures on large system sizes by a factor of up to 2x. Improving the scalability of the algorithm and exploring Hybrid algorithm on shared memory systems are useful avenues for future work.

References

1. Cohen, J.: Trusses: cohesive subgraphs for social network analysis. Technical report, National Security Agency (2008)
2. Saito, K., Yamada, T., Kazama, K.: Extracting communities from complex networks by the k-dense method. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **91**(11), 3304–3311 (2008)
3. Alvarez-Hamelin, J., Dall'Asta, L., Barrat, A., Vespignani, A.: Large scale networks fingerprinting and visualization using the k-core decomposition. In: NIPS (2005)
4. Huang, X., Lakshmanan, L., Yu, J., Cheng, H.: Approximate closest community search in networks. Proc. VLDB Endow. **9**(4), 276–287 (2015)
5. Seidman, S.: Network structure and minimum degree. Soc. Netw. **5**(3), 269–287 (1983)
6. Sariyuce, A., Seshadhri, C., Pinar, A., Catalyurek, U.: Finding the hierarchy of dense subgraphs using nucleus decompositions. In: WWW (2015)
7. Wang, J., Cheng, J.: Truss decomposition in massive networks. Proc. VLDB Endow. **5**(9), 812–823 (2012)
8. Rossi, R.A.: Fast triangle core decomposition for mining large graphs. In: Tseng, V.S., Ho, T.B., Zhou, Z.-H., Chen, A.L.P., Kao, H.-Y. (eds.) PAKDD 2014. LNCS (LNAI), vol. 8443, pp. 310–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06608-0_26
9. Smith, S., Liu, X., Ahmed, N., Tom, A., Petrini, F., Karypis, G.: Truss decompositions on shared-memory parallel systems. In: HPEC (2017)
10. Kabir, H., Madduri, K.: Shared-memory graph truss decomposition. In: HiPC (2017)
11. Kabir, H., Madduri, K.: Parallel k-truss decomposition on multicore systems. In: HPEC (2017)
12. Voegele, C., Lu, Y., Pai, S., Pingali, K.: Parallel triangle counting and k-truss identification using graph-centric methods. In: HPEC (2017)
13. Green, O., et al.: Quickly finding a truss in a haystack. In: HPEC (2017)

14. Zhang, Y., Parthasarathy, S.: Extracting analyzing and visualizing triangle k-core motifs within networks. In: ICDE (2012)
15. Chen, P., Chou, C., Chen, M.: Distributed algorithms for k-truss decomposition. In: IEEE International Conference on Big Data (2014)
16. Cohen, J.: Graph twiddling in a MapReduce world. Comput. Sci. Eng. **11**(4), 29–41 (2009)
17. Shao, Y., Chen, L., Cui, B.: Efficient cohesive subgraphs detection in parallel. In: SIGMOD (2014)
18. Kolda, T., Pinar, A., Plantenga, T., Seshadhri, C., Task, C.: Counting triangles in massive graphs with MapReduce. SIAM J. Sci. Comput. **36**(5), S48–S77 (2014)
19. Meyer, U., Sandersr, P.: Δ-stepping: a parallelizable shortest path algorithm. J. Algorithms **49**(1), 114–152 (2003)
20. Boldi, P., Vigna, S.: The webgraph framework i: compression techniques. In: WWW (2004)