High-Performance Truss Analytics in Arkouda

Zhihui Du, Joseph Patchett, Oliver Alvarado Rodriguez, Fuhuan Li and David A. Bader
Department of Data Science, New Jersey Institute of Technology, Newark, USA
{zhihui.du,jtp47,oaa9,fl28,bader}@njit.edu

***Abstract*—In graph analytics, a truss is a cohesive subgraph based on the number of triangles supporting each edge. It is widely used for community detection applications such as social networks and security analysis, and the performance of truss analytics highly depends on its triangle counting method. This paper proposes a novel triangle counting kernel named Minimum Search (MS). Minimum Search can select two smaller adjacency lists out of three and uses fine-grained parallelism to improve the performance of triangle counting. Then, two basic algorithms, MS-based triangle counting, and MS-based support updating are developed. Based on the novel triangle counting kernel and the two basic algorithms above, three fundamental parallel truss analytics algorithms are designed and implemented to enable different kinds of graph truss analysis. These truss algorithms include an optimized K-Truss algorithm, a Max-Truss algorithm, and a Truss Decomposition algorithm. Moreover, all proposed algorithms have been implemented in the parallel language Chapel and integrated into an open-source framework, Arkouda. Through Arkouda, data scientists can efficiently conduct graph analysis through an easy-to-use Python interface and handle large-scale graph data in powerful back-end computing resources. Experimental results show that the proposed methods can significantly improve the performance of truss analysis on real-world graphs compared with the existing and widely adopted list intersection-based method. The implemented code is publicly available from GitHub (https://github.com/Bears-R-Us/arkouda-njit).**

***Index Terms*—K-Truss, Triangle Counting, Graph Analytics, Parallel Algorithm**

I. Introduction

K-Trusses [10] have been widely used to discover close relationships in a graph and are more rigorous than *k-cores* (where all the vertices have a degree at least k in a subgraph) but less stringent than *k-cliques* (where all the vertices are connected pairwise in a subgraph). The k-clique decision problem is NP-complete, but *K-Trusses* can be computed in polynomial time, making them more practical in large graph analysis. However, the increasing size of real-world graphs has become a great challenge for *K-Truss* identification.

Furthermore, *Max-Truss* analysis [11], which aims to identify the largest *K-Truss* of a graph, is more compute-intensive than general *K-Truss* analysis. The more comprehensive *Truss-Decomposition* [28] will generate all trusses with different K values and it is the costliest truss-based algorithm. Triangle counting and support (here, support is the number of triangles using a given edge) updating are the major costs for truss analysis. Currently, list intersection-based triangle counting is the most popular and widely used method for truss analytics. However, the list intersection-based triangle counting method [11] cannot take advantage of the properties of real-world graphs to improve its performance. In this paper, we propose a novel *Minimum Search* based triangle counting and support updating method to use in the three typical truss analytics algorithms that can achieve much better performance, especially for real-world graphs. Moreover, all the truss analytics algorithms have been integrated into an open-source framework Arkouda [22], [23] to support exploratory data analysis (EDA) [5], [15], [20] at scale.

EDA has become a critical method for data scientists to discover the value of data quickly. Unfortunately, most EDA tools, which often run on laptops or common personal computers, cannot handle large data efficiently, let alone produce highly productive analysis results. Developing efficient *Truss* analytics algorithms, software tools, and environments that can take advantage of the latest hardware resources through high-level and user-friendly interfaces is essential. Such algorithms, tools, and environments can enable EDA users and data scientists to conduct their analysis more productively.

Arkouda is an EDA framework under early development that combines Python's productivity at the front end with the high-performance computing capability of Chapel [9] at the back end. This framework is perfect for achieving productive large graph analysis on powerful computing resources with an easy-to-use and popular Python interface. In this work, we develop and integrate the proposed parallel *Truss* analytics algorithms into Arkouda so that data scientists can take advantage of Python and use their laptops to conduct real-world graph analysis on large compute platforms (including clusters) productively.

The major contributions of this paper are as follows.

1) A novel *Minimum Search*-based triangle counting kernel is developed. The time complexity analysis exposes why it is more efficient compared with the widely used *List Intersection* method.
2) All our *Truss* analytics algorithms based on the *Minimum Search* kernel have been implemented and integrated into an open-source framework Arkouda. Through Arkouda, data scientists can conduct interactive *Truss* analytics at scale using Python with a high-performance guarantee with powerful back-end support.
3) Extensive experimental results show that the proposed performance optimization methods can achieve significant speedup compared with the widely used *List Intersection* method, especially on real-world graphs.

2640-0316/22/$31.00 ©2022 IEEE
DOI 10.1109/HiPC56025.2022.00026

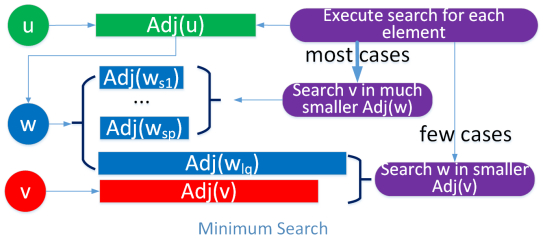

Fig. 1: Basic idea of the novel minimum search method.

II. Novel Triangle Counting Kernel

A. Minimum Search for Triangle Counting

Given edges $e = \langle u, v\rangle \in E$, let the adjacency lists of u and v be Adj_u and Adj_v; then, the number of triangles including e should be $|Adj_u \cap Adj_v|$. This is the formula of the widely used *List Intersection* method [11] in *K-Truss* analysis.

If Adj_u and Adj_v are sorted, then the execution time of the efficient *Intersect Path* method [17] to implement *List Intersection* can be $|Adj_u| + |Adj_v|$. Although *Intersect Path* can be executed in parallel, the data distribution in the two adjacency lists can significantly affect the parallelism obtained.

Intersect Path uses only two adjacency lists of the two vertices in the given edge to find triangles based on the intersection of the adjacency lists. We propose a novel triangle counting kernel called *Minimum Search (MS)*. *MS* can find each triangle containing the given edge with much fewer operations because it considers all three adjacency lists and only executes a search in the smaller adjacency list. The basic idea of *MS* is shown in Fig. 1.

Given edge $e = \langle u, v\rangle$, we assume that $|Adj(u)| \leq |Adj(v)|$. $\forall w \in Adj(u)$, we will use $|Adj(u)|$ parallel threads to check if u, v, w can form a triangle. Furthermore, if $|Adj(w)| < |Adj(v)|$, then we will check if $v \in Adj(w)$. Otherwise, we will check if $w \in Adj(v)$. This means that we only select the smaller adjacency list for the triangle checking. However, the existing *List Intersection* methods use only two adjacency lists, $Adj(u)$ and $Adj(v)$ for triangle checking even if $Adj(v)$ is very large compared with $Adj(w)$. So, the potential time savings from a much smaller adjacency list $Adj(w)$ are completely ignored. The highly-skewed degree distributions in real-world graphs mean that most of the $Adj(w)$ will be very small. So, our *MS* method can save significant time because it only searches in the smaller adjacency list.

B. Time Complexity Analysis

By analyzing its time complexity, we can show that the proposed *Minimum Search* method can achieve a faster execution time than the existing *List Intersection* method.

Let h (l) be u or v which has more (less) adjacent vertices. Let Adj_h (Adj_l) be Adj_u or Adj_v that has more (less) elements. $\forall w \in Adj_l$, let Adj_w be the adjacency list of w. The proposed *Minimum Search* method directly checks if there is a third edge $\langle w, h\rangle$ that can close the wedge $\langle l, h\rangle$ and $\langle l, w\rangle$ to form a triangle. We assume that all vertices' adjacency lists are sorted in the preprocessing stage. Then, the execution time or the number of search operations of the *Minimum Search* method will be $log_2(min(|Adj_w|, |Adj_h|))$. For the existing *List Intersection* method, if the same parallel method is employed, its execution time should be $log_2(|Adj_h|)$ because it only checks w in $Adj(v)$. The larger the difference between $|Adj_w|$ and $|Adj_h|$, the greater the number of operations that can be saved. For the proposed *Minimum Search* method, the total time to search all the triangles containing given edge $\langle u, v\rangle$ in parallel can be calculated as in Eq.1.

$$\max_{w \in Adj_l} log_2(min(|Adj_w|, |Adj_h|)) \tag{1}$$

For example, if $|Adj_l| = 4$, $\forall w \in adj_l, |Adj_w| \leq 8$ and $|Adj_h| = 1024$, the existing *List Intersection* method (employing the same fine-grained parallel method as ours) will need four parallel threads and each thread will execute $\lceil log_2 1024 \rceil = 10$ operations to search the triangles containing given edge $\langle u, v\rangle$. Our novel method will take $\lceil log_2 8 \rceil = 3$ search operations with the same number of four threads. It is less than the standard *List Intersection*'s parallel execution time.

If fine-grained parallelism is not supported, the total time of our sequential *Minimum Search* to find all triangles containing given edge $\langle u, v\rangle$ can be calculated as in Eq.2.

$$\sum_{w \in Adj_l} log_2(min(|Adj_w|, |Adj_h|)) \tag{2}$$

Compared with the efficient *Intersect Path* method [16], its sequential execution time is

$$(|Adj_l| + |Adj_h|) \tag{3}$$

Based on Eq.2 and Eq.3, even under the worst case, $\forall w \in Adj_l$, if $|Adj(w)| \geq |Adj(h)|$ and $\frac{|Adj(h)|}{|Adj(l)|} > (log_2|Adj(h)| - 1)$, we still have *T(Sequential Minimum Search) < T(Sequential Intersect Path)*. Fortunately, this condition can be met with a high possibility for real-world graphs whose degrees follow power-law distributions.

For example, let $|Adj(u)| = 8$ and $|Adj(v)| = 128$. Then, *Intersect Path* method will need $(8 + 128 = 136)$ operations. But our *Minimum Search* will need at most $8 \times log_2(128) = 56$ operations even if $|Adj(w)| > 128$ for $\forall w \in Adj_l$. The experimental results (see section V-B) also show that *T(Sequential Minimum Search) < T(Sequential Intersect Path)* for real-world graphs. This is a "surprising" result and confirms that our *Minimum Search*-based kernel can take advantage of real-world graphs' properties to improve performance.

Briefly, the proposed *MS* kernel has the following advantages: (1) Exploit fine-grained parallelism; (2) Avoid search in the largest adjacency list; (3) Leverage skewed degrees of real-world graphs.

C. MS based Triangle Counting

In this section, we will use the novel *Minimum Search* kernel to build the triangle counting algorithm that can be seen in Alg.

1. The algorithm is given in the Chapel-based pseudocode. All the algorithms presented in this work can support multiple locales[1]. In the Chapel parallel language, the locale type refers to a unit of the machine resources on which a program is running.

In Alg. 1, we use an atomic array $AtoSup$ to store the support of each edge in a given graph G to avoid write race. Array $EDel$ is used to mark the state of each edge e. $EDel[e]$=*-1* means that edge e has not been deleted. $EDel[e]$=*k* means that edge e has been deleted and its trussness is k. $EDel[e]$=*1-k* means that edge e has just been removed and we are updating the support of other edges that can form a triangle with e in the next step.

In lines 2 to 19 we show how to enable parallel execution across different locales. On each locale, we calculate the support of each edge e that has not been deleted and is local to the current locale. From lines 6 to 10, we search vertex v in the smaller adjacency list $Adj(w)$, where $w \in Adj(u)$. Otherwise, from lines 11 to 15, we search vertex w in the smaller adjacency list $Adj(v)$. After getting all the number of triangles $Count$ that contain edge e, we assign it to $AtoSup[e]$ in line 17.

To summarize, our edge-based method can achieve load balance by assigning edges evenly to different locales. For any vertex w that may form a triangle with vertices u and v, we only use the smaller adjacency lists and avoid searching in the largest adjacency list. In this way, our *MS*-based method can significantly improve the performance of triangle counting.

Algorithm 1: *Minimum Search* based Triangle Counting

```
1  TriangleCounting(G, EDel, AtoSup)
2  coforall loc in Locales do
3      forall (undeleted edge e = ⟨u, v⟩ ∈ E) && (e is
         local) do
           /* We assume that |Adj(u)| < |Adj(v)|   */
4          var Count:int=0;
5          forall w ∈ Adj(u) with (+ reduce Count ) do
6              if (|Adj(w)| < |Adj(v)|) then
7                  if (v ∈ Adj(w)) then
8                      Count++;
9                  end
10             end
11             else
12                 if (w ∈ Adj(v)) then
13                     Count++;
14                 end
15             end
16         end
17         AtoSup[e].write(Count);
18     end
19 end
```

D. MS based Support Updating

If some edges are removed from a graph, updating the number of triangles of affected edges instead of recalculating the number of triangles from scratch can often save significant time. So, the support updating algorithm is critical for improving the performance of truss analytics.

Say that $\forall e_1, e_2$, and $e_3 \in E$ can form a triangle. If, for example, e_1 is deleted, then the number of triangles of e_2 and e_3 will be affected and they are the "affected edges" of e_1. We need to reduce the supports of e_2 and e_3 after e_1 is removed. However, our support updating algorithm should avoid updating the same undeleted edge e_3 twice If both e_1 and e_2 are removed. At the same time, one unremoved edge may be affected by multiple removed edges in different triangles. So, we use an atomic subtraction operation to reduce the support of the unremoved edge to avoid a write race. The support updating algorithm based on our novel *MS* kernel is given in Alg. 2.

Support updating will start from the edges that have just been removed. We employ a high-level Chapel data structure *DistributedBag*[2] to keep the edges that have just been removed. The structures of Alg. 1 and Alg. 2 have some similarity. However, there are two differences. (1) Instead of searching from undeleted edges, we will search from just deleted edges. (2) When a triangle that contains the given edge is found, the support of the corresponding edge will be reduced instead of increasing the number of triangles. To avoid reducing the support twice by two different removed edges in the same triangle, we only allow the deleted edge with a smaller edge ID to execute the update. In addition, we employ atomic operations to update the support at lines 8, 9, 13, and 16 to avoid write race because several deleted edges may update the support of the same edge simultaneously.

Assume all edges that have just been removed are put into a *DistributedBag* named *JustDelEBag*. On each locale, we will search from the edges in *JustDelEBag*, which are on the current locale. Each locale will execute in parallel the search from different edges (line 3). We use e_1 to stand for the edge in the just-removed bag. e_2 and e_3 stand for the other two edges (they can be the just removed edges in *JustDelEBag*) that may form a triangle with e_1. Once e_2 and e_3 are found, their supports will be reduced by one if both are undeleted edges (lines from 7 to 10). Otherwise, if one of them is a deleted edge, we will reduce its support only by the edge whose ID is smaller than another deleted edge to avoid double updates on the same edge's support (lines from 12 to 14 and lines from 15 to 17).

Based on the *MS* based triangle counting and *MS* based support updating algorithms, we can build our *K-Truss* algorithm, *Max-Truss* algorithm and *Truss Decomposition* algorithm efficiently.

III. Truss Analytics Algorithm Design

A. Problem Description

1) Notation: A graph, $G = (V, E)$, is comprised of a vertex set V and an edge set E. We use $\triangle(e, G)$ to express the set of all triangles including edge $e = \langle u, v\rangle$ in the graph G. The

[1]https://chapel-lang.org/docs/language/spec/locales.html

[2]https://chapel-lang.org/docs/modules/packages/DistributedBag.html

Algorithm 2: *Minimum Search* based Support Updating

```
1  SupportUpdate(G, EDel, JustDelEBag, AtoSup)
2  coforall loc in Locales do
3      forall (e1 = ⟨u, v⟩ ∈ JustDelEBag) && (e is local) do
           /* We assume that |Adj(u)| < |Adj(v)|  */
4          forall (e2 = ⟨u, w⟩, w ∈ Adj(u) − {v}) &&
             (EDel[e2] < 0) do
5              Search e3 = ⟨v, w⟩ or e3 = ⟨w, v⟩;
6              if (e3 exists) then
7                  if (e2 and e3 are undelted) then
8                      AtoSup[e2].sub(1);
9                      AtoSup[e3].sub(1);
10                 end
11                 else
12                     if (e2 is undeleted) &&(e1 < e3) then
13                         AtoSup[e2].sub(1);
14                     end
15                     if (e3 is undeleted) &&(e1 < e2) then
16                         AtoSup[e3].sub(1);
17                     end
18                 end
19             end
20         end
21     end
22 end
```

support of e, which means the number of triangles including edge $e = \langle u, v\rangle$ in G, is expressed as $sup(e, G) = |\triangle(e, G)|$. In truss analysis, we ignore the direction of the edges in a graph G or, simply, G is an undirected graph.

Definition 1. *K-Truss: Given a graph G, its K-Truss is defined as the maximal non-empty subgraph $SubG = \langle SubV, SubE\rangle$ such that $\forall e \in SubE \subseteq E$, we will have $sup(e, SubG) \geq$ K-2, where K is an integer and $K \geq 2$. A large K may cause that no any subgraph can meet the requirement.*

Definition 2. *Max-Truss: Given a graph G, its Max-Truss is defined as the K-Truss that has the maximum value of K among all the K-Trusses of G.*

Definition 3. *Truss Decomposition: Given a graph $G = (V, E)$, the Truss Decomposition of a graph G is assigning each edge with its Truss Value. $\forall e \in E$, the Truss Value or Trussness of e is defined as the maximum K value of all K-Trusses that includes e. It can be expressed as $truss(e, G)$ and we have $truss(e, G) \leq sup(e, G)$.*

2) MaxK Analysis: Based on the above definition, the minimum value of K is 2. We use $MaxK$ to denote the maximum K value for a graph G and the corresponding subgraph $MaxKG = \langle MaxV, MaxE\rangle$.

Theorem 1 (*MaxK Inequality*)**.** *Given $MaxK$ of a graph G, we have $\forall v \in MaxV, MaxK \leq degree(v)$-1 and $MaxK \leq |MaxV|$.*

Proof. Based on the definition of *Max-Truss*, $\forall e = \langle v_1, v_2\rangle \in MaxE$, we will have $sup(e, MaxKG) \geq MaxK$-2. This definition requires that $MaxK \leq degree(v_1)$-1 and $MaxK \leq degree(v_2)$-1. Otherwise, it will be impossible to form $MaxK$-2 triangles with e. $\forall v \in MaxV$, it must be the vertex of one $e \in MaxE$. So, we have $MaxK \leq degree(v)$-1. This definition also requires that the total number of vertices in $MaxV$ should meet $|MaxV| \geq MaxK$. If we assume that $|MaxV| < MaxK$, then $\forall e \in MaxE$, we cannot find $MaxK$-2 different vertices to form triangles with e. This is a contradiction with the fact $sup(e, MaxKG) \geq MaxK$-2. □

Based on Theorem 1, we can estimate an upper bound k_u of $MaxK$ for a given graph using the following method. First, we sort the vertices in decreasing order based on their degrees. Second, we add the vertices into a set $VSet$ one by one. Let $Dmin = min\{degree(v)|v \in VSet\}$ and we let $k_u = max\{x|x = |VSet| \wedge x \leq Dmin - 1\}$ for all possible $VSet$, we will have $MaxK \leq k_u$. In this way, we may use k_u to set the upper bound of $MaxK$.

3) Double Index Data Structure: This paper focuses on sparse graphs that can model a wide range of real-world applications such as social networks, bioinformatics, and cybersecurity. The CSR (Compressed Sparse Row) data structure is widely used to express sparse graphs. However, it is a vertex-centric data structure and cannot directly support edge ID-based search. At the same time, real-world graphs are often highly skewed in vertices' degrees [2], [26]. The vertex-based graph partition will often cause an unbalanced workload. So, a compact and efficient Double-Index (DI) sparse graph data structure (edge index and vertex index) [14] is employed in this research to support our *K-Truss* analysis. The DI data structure enables quick edge-based and vertex-based search by taking advantage of its edge index and vertex index arrays. At the same time, the edge index arrays can be used to partition a graph's edges evenly to achieve load balance for edge-based graph algorithms. All these features provide the foundation to build quick triangle search-based truss analytics methods and tools.

B. Naïve K-Truss Parallel Algorithm

In this section, we will first introduce the naïve method to show the basic idea of *K-Truss* analysis.

Algorithm 3: Naïve *K-Truss* Parallel Algorithm

```
1  NaiveKTruss(G, k)
2  AtoSup ← 0 and EDel ← -1
3  while (there is any edge can be deleted) do
4      TriangleCounting(G, EDel, AtoSup)
5      coforall loc in Locales do
6          forall (e = ⟨u, v⟩ ∈ E) && (e is local) do
7              if (EDel[e]==-1) &&
                 (AtoSup[e].read() < k-2) then
8                  EDel[e]=k-1
9              end
10         end
11     end
12 end
13 return EDel
```

The *DI* sparse graph data structure can support addressing in both vertex and edge [14] quickly. We employ *DI* to develop a naïve but distributed parallel framework for *K-Truss* algorithm that can be easily implemented in Chapel first.

Peeling [10] is a simple but very efficient *K-Truss* subgraph generation method. It removes the edges whose number of triangles is less than K-2 step by step, like peeling an onion. We propose a naïve version of this method in Alg. 3.

Our naïve algorithm can run on distributed memory systems to take advantage of multiple computing resources to handle large graphs. At the same time, in each shared-memory multicore/SMP node, the triangle counting and the checking for different edges on the current locale can also be executed in parallel. In line 4, we call the *MS* based triangle counting to count the number of triangles of each undeleted edge in parallel. From lines 5 to 11, we check each undeleted edge and mark the edges whose supports are less than $(k-2)$ with $(k-1)$, which means that it has been deleted because it cannot meet the $(k-2)$ requirement.

The naïve *K-Truss* algorithm shows how we can exploit parallelism and employ our novel triangle search kernel for *K-Truss* analysis.

C. Optimized K-Truss Parallel Algorithm

The naïve *K-Truss* algorithm is simple and easy to implement. Under most scenarios, it cannot achieve high performance even though it has a natural parallel framework. The reason for its low performance is that it will recalculate the number of triangles in each iteration. The more iterations it has, the more unnecessary triangle counting operations will be executed.

The primary optimization method of the optimized *K-Truss* algorithm is parallel searching the affected edges to avoid repeat triangle counting [3], [7], [12], [16], [21]. After the supports of all affected edges have been updated, the unremoved edges whose supports are less than $(k-2)$ will also be removed. All the newly removed edges will be used to parallel search new affected edges until no affected edges can be removed. This optimization can avoid repeat triangle counting from scratch. So, it can significantly reduce the total number of operations. Here, we combine the *MS* based triangle searching and *MS* based support updating together to build the optimized *K-Truss* algorithm, which is given in Alg. 4.

In Alg. 4, we first call the *MS*-based triangle counting to calculate the number of triangles of all the edges (line 4), just as we do in the naïve method. The significant difference is the second part. We will continuously remove the affected edges whose supports are less than $(k-2)$ until no such edges can be found (lines 13 to 29). In lines 5 to 12, we check the supports of all the edges in parallel and put the edges whose supports are less than $(k-2)$ in the $JustDelEBag$. At the same time, the corresponding edges are marked as $(1-k)$ which means "just removed". In line 14, we call the *MS* based support updating algorithm to change the support of affected edges based on the just removed edges in $JustDelEBag$. In lines 16 to 20, we mark the just removed edges as removed

Algorithm 4: Optimized *K-Truss* Parallel Algorithm

```
1  OptKTruss(G, k)
2  var JustDelEBag= new DistBag(int,Locales);
3  AtoSup ← 0 and EDel ← -1
4  TriangleCounting(G, EDel, AtoSup)
5  coforall loc in Locales do
6      forall (e ∈ E) && (e is local) do
7          if (EDel[e]==-1) && (AtoSup[e].read() < k-2)
             then
8              EDel[e]=1-k
9              JustDelEBag.add(e);
10         end
11     end
12 end
13 while (JustDelEBag.getSize() > 0 ) do
14     SupportUpdate(G, EDel, JustDelEBag, AtoSup)
15     coforall loc in Locales do
16         forall (e ∈ JustDelEBag) && (e is local) do
17             if (EDel[e]==1-k) then
18                 EDel[e]=k-1
19             end
20         end
21         JustDelEBag.clear();
22         forall (e ∈ E) && (e is local) do
23             if (EDel[e]==-1) &&
                 (AtoSup[e].read() < k-2) then
24                 EDel[e]=1-k
25                 JustDelEBag.add(e);
26             end
27         end
28     end
29 end
30 return EDel
```

edges by changing their values from $(1-k)$ to $(k-1)$. In line 21, we clear all the edges in $JustDelEBag$. Then, we search for new edges that can be removed after the support updating (lines 22 to 27). They are marked as value $(1-k)$ and added into $JustDelEBag$ for the next iteration to update support of new affected edges.

In summary, our optimized *K-Truss* algorithm can take advantage of the proposed *MS* based triangle counting and support updating methods to quickly find the triangles and avoid recounting the number of triangles from scratch.

D. Max-Truss Parallel Algorithm

We develop a binary search-like *Max-Truss* algorithm to identify the maximum truss and its K value quickly. We develop our *Max-Truss* algorithm based on the proposed optimized parallel *K-Truss* algorithm. To optimize the performance, we will reuse the support array $AtoSup$ and the edge state array $EDel$ to avoid repeat triangle counting operations in the optimized parallel *K-Truss* algorithm.

According to the discussion in section III-A, we can first calculate the upper bound k_u of the maximum *K-Truss* value. So, we only need to check the maximum k value in range $[3..k_u]$ that will not delete all the edges in a graph. That k will be the maximum *K-Truss* value of the given graph. In Alg. 5 we give the description of our *Max-Truss* parallel algorithm.

In line 2 we initialize the range of maximum *K-Truss* search value k_l and k_u. Based on the feature of *K-Truss* search, we have the inequality $k_l \leq MaxK \leq k_u$. First, we call the optimized *K-Truss* method in line 5 with k_l and then check if k_l can remove all the edges in G in line 6. If it can, we know that $(k_l = 2)$ is the maximum k value. Otherwise, we call the $MaxSearch$ procedure at line 10 to return the maximum k value with given range $[(k_l$+2)..$k_u]$ using the existing edge array $EDel$ and support array $AtoSup$. In the $MaxSearch$ function, all searches will be based on edge array $EDel$ and support array $AtoSup$ instead of recalculating them from scratch.

In lines 12 to 31, we define the $MaxSearch$ function. To avoid recalculating the support array $AtoSup$ and to allow sharing of the edge array $EDel$, we modify the *OptKTruss* algorithm in Alg. 4 by removing the triangle counting operations in line 4 and rename the new algorithm as *MaxOptKTruss*. We add edge array $EDel$ and support array $AtoSup$ as the parameters of *MaxOptKTruss* to avoid recomputing them.

There are two differences with the binary search method. First, we need to store the existing edge array $EDel$ and support array $AtoSup$ in line 22 before a new k_{mid} checking in line 23. The reason is that if the new k_{mid} will remove all the edges, we need to recover them in line 28 to recursively search in a new range whose upper bound cannot be larger than the k_{mid} value (lines from 28 to 30). Second, k_{mid} is not the exact middle value of k_l and k_u but close to k_l. This is because large k_{mid} can cause additional overhead if it will remove all the edges. Our experimental results also confirm that this optimization can really improve performance.

Based on our optimized *K-Truss* parallel algorithm, the *Truss Decomposition* procedure is straightforward. We just need to increase the value of k based on the minimum support of all undeleted edges in the remaining subgraph. If the minimum support is s, then the new $k = (s+2)$ until all edges have been removed. So, we ignore the detailed description on *Truss Decomposition* algorithm here.

Algorithm 5: *Max-Truss* Parallel Algorithm

```
1  MaxTruss(G)
2  Let k_l = 2 and set k_u based on the method in sec.III-A
3  EDel = −1
4  Declare AtoSup as an atomic int array
5  OptKTruss(G, k_l + 1)
6  if (All edges have been deleted) then
7  |  return (k_l,EDel)
8  end
9  else
10 |  return MaxSearch(G, k_l+2, k_u, EDel, AtoSup)
11 end
12 proc MaxSearch(G, k_l, k_u, EDel, AtoSup)
13 MaxOptKTruss(G, k_l, EDel, AtoSup)
14 if (All edges have been deleted) then
15 |  return (k_l-1, EDel)
16 end
17 else
18 |  if k_l ≥ k_u then
19 |  |  return (k_l, EDel)
20 |  end
21 |  k_mid = (k_l+1)+(k_u − k_l)/3
22 |  (BakEDel, BakAtoSup) ← (EDel, AtoSup)
23 |  MaxOptKTruss(G, k_mid, EDel, AtoSup)
24 |  if (there are undeleted edges) then
25 |  |  return
   |  |    MaxSearch(G, k_mid+1, k_u, EDel, AtoSup)
26 |  end
27 |  else
28 |  |  (EDel, AtoSup) ← (BakEDel, BakAtoSup)
29 |  |  return
   |  |    MaxSearch(G, k_l+1, k_mid-1, EDel, AtoSup)
30 |  end
31 end
```

IV. Integration with Arkouda

Arkouda[3] is an open-source framework that allows data scientists to conduct productive data analytics from their laptops by transferring the burden of high-performance computing to a back-end server. Arkouda contains three major components: an interactive Python front-end, a ZeroMQ middleware, and a Chapel back-end. We need to extend the front-end and the back-end to integrate our algorithms into Arkouda.

After implementing the kernels into Chapel with our novel data structures and algorithms, we need to follow Arkouda's integration rules to make the new functionality work well to create an end-to-end response from Chapel to Python.

A. Easy Python Interface at Front-End

All communication between the Python front-end and Chapel back-end has been implemented in the Arkouda framework. Based on the Arkouda framework, we define one Python function *KTruss(graph, k)* for all truss analytics methods.

For the truss function call, if the value of parameter k is larger than 0, it means that we will execute the *K-Truss* analysis and return the maximum subgraph that every edge's support is not less than $(k-2)$. We use the same function for *Max-Truss* or *Truss Decomposition* analysis. If $(k$==*-1*$)$, we will call the *Max-Truss* algorithm at the back-end. If $(k$==*-2*$)$, we will call the *Truss Decomposition* algorithm.

B. Optimized Chapel Implementation at Back-End

The major integration work is at the Chapel back end. The Arkouda framework will transfer a front-end Python function call to the back-end Chapel implementation. The Chapel execution results will also be transferred back to the Python function as return results. So, at the back end, we only need to implement the different truss analytics algorithms as different Chapel functions.

The back-end accepts the command's name given by the Python front-end, a payload message, and a symbol table name where our data will be housed from the Chapel back-end. The payload is parsed to extract the name of the Chapel graph class that houses our graph data, and then using the name, we extract the data from the symbol table and then work with it to run our different algorithms. We develop the *kTruss-*

[3]https://github.com/Bears-R-Us/arkouda

MinSearch() Chapel function to implement our optimized parallel minimum search based *K-Truss* algorithm Alg. 4. The *MaxTrussMinSearch()* function is developed to implement our *Max-Truss* algorithm Alg. 5. The *TrussDecoMinSearch()* function is developed to implement our *Truss Decomposition* algorithm. The back-end will call different functions based on the value of parameter k.

C. Tools for Performance Comparison and Optimization

We develop the following algorithms to compare the performance of the proposed *Minimum Search* kernel-based truss analytics algorithms with existing methods. *Intersect Path* is an efficient implementation of *List Intersection* method. We implement the *Naïve Intersect Path* algorithm (*NaïveIP*) as the baseline. At the same time, we also develop the following algorithms for comparison. The optimized *Intersect Path* algorithm (*IP*) shows the effect of removing repeat triangle counting from scratch. The optimized *Minimum Search* algorithm (*MS-2*) only uses two adjacency lists to show the performance loss when we cannot take advantage of three adjacency lists. *Direct Graph*-based triangle counting is an important optimization method to update the supports of three edges in one triangle search to avoid searching the same triangle three times. We develop such a comparison method and name it *Dir*. All the five different algorithms are implemented in the *K-Truss*, *Max-Truss* and *Truss Decomposition* analyses.

V. Experiments

A. Dataset Description

Our datasets are chosen from publicly available synthetic and real-world datasets. The SuitSparse Matrix Collection[4] and the MIT GraphChallenge graph datasets[5] are used in selecting our test graphs. A combination of real-world and synthetic graphs are selected to highlight the performance of our optimized *Minimum Search* kernel-based methods. These graphs can be found in Table I. The real-world graphs can be either weighted or unweighted. Real-world graphs have degree distributions that follow a power-law distribution, while sparse synthetic graphs tend to follow a normal distribution. The two kinds of synthetic graphs used are Delaunay and random geometric graphs (Rgg) from the DIMACS10 [4]. Delaunay graphs are composed of Delaunay triangulations of random points in the plane, whereas Rgg graphs are just random points in the unit square. Both have multiple triangles, making them interesting graphs to benchmark truss problems.

B. Performance Results

Experiments were done on two platforms. Platform 1 (P1) is a CentOS Linux release 7.9.2009 (Core) high-performance server with 2 x Intel Xeon E5-2650 v3 @ 2.30GHz CPUs with ten cores per CPU. The server has an amount of 512GB of RAM. Platform 2 (P2) is a GridOS 18.04 Linux cluster. Our experiments are on two computing nodes composed of Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz (total 96 cores) and 128GB of memory.

[4]https://sparse.tamu.edu/

[5]https://graphchallenge.mit.edu/data-sets

TABLE I: Dataset Descriptions

	Graph Name	Graph ID	Number Edges	Number Vertices	Max K
Real-World Graph	ca-GrQc	1	14484	5242	44
	ca-HepTh	2	25973	9877	32
	as-caida20071105	3	53381	26475	16
	facebook_combined	4	88234	4039	97
	ca-CondMat	5	93439	23133	26
	ca-HepPh	6	118489	12008	239
	email-Enron	7	183831	36692	22
	ca-AstroPh	8	198050	18772	57
	loc-brightkite_edges	9	214078	58228	43
	soc-Epinions1	10	405740	75879	33
	amazon0601	11	2443408	403394	11
	com-Youtube	12	2987624	1134890	19
Synthetic Graph	delaunay_n20	13	3145686	1048576	4
	delaunay_n21	14	6291408	2097152	4
	delaunay_n22	15	12582869	4194304	4
	delaunay_n23	16	25165784	8388608	4
	rgg_n _2_21	17	14487995	2097152	19
	rgg_n _2_22	18	30359198	4194304	20
	rgg_n _2_23	19	63501393	8388608	21
	rgg_n _2_24	20	132557200	16777216	21

We first provide the experimental results on Platform 1. The speedups of our optimized Truss analytics algorithms based on *Minimum Search* compared with the *List Intersection* are given in Fig. 2. For all the real-world graphs, the average speedups of our *Minimum Search* based algorithms are 4.49 (*K-Truss*), 3.54 (*Max-Truss*) and 3.60 (*Truss Decomposition*). The maximum speedups are 17.88, 14.31 and 14.88 achieved on the highly skewed graph "com-Youtube" with a maximum degree as high as 28754 but an average degree as low as 5. However, for synthetic graphs, their performance is very close to each other because their degree distributions are not as skewed as those of real-world graphs. The results show the significant advantage of our *Minimum Search* based algorithms compared with the widely used *List Intersection* based algorithms on real-world graphs.

Then, we show the effect of different optimization methods compared with the *Naïve IP* implementation in Fig. 3, Fig. 4 and Fig. 5. In all the graphs, "IP" is the optimized *Intersect Path* algorithm's results. "IP" employs the affected edge search method instead of recalculating triangles from scratch. "MS-2" shows the difference if we limit our optimized method that can only use given two adjacency lists instead of three. "MS" is our optimized *Minimum Search* based algorithm. The directed graph method is also very popular for optimizing the performance of undirected graphs. "Dir" is the directed graph method supported by the undirected *DI* data structure.

The experimental results of the *K-Truss* algorithms in Fig. 3 show that: (1) All undirected graph-based optimized methods (*IP, MS-2*, and *MS*) can achieve much better performance than that of the *Naïve IP* method on both real-world graphs and synthetic graphs. The results show that repeat triangle counting is very expensive in *K-Truss* analysis. However, the directed graph-based *Dir* method can achieve very limited speedup, or even worse than the *Naïve IP* method. The reason is that the *Dir* method only allows one edge to update the supports of the three edges that can form a triangle under the edge-based parallel triangle search scenario. This means that $\frac{2}{3}$ of

the parallel threads cannot update their supports directly. This low resource utilization will reduce the performance of the directed graph-based method. Under the vertex-based parallel triangle search framework, the directed graph-based triangle counting method can often achieve good performance by avoiding multiple triangle searches. However, it is not suitable for the edge-based parallel triangle search framework.

(2) Our optimized *MS* algorithm can achieve the best performance for real-world graphs. It is much better than the *IP* algorithm. The highest speedup is 89.07 (achieved on the real-world graph "com-youtube"), and the average speedup is 22.88 for all the testing graphs. In Alg. 2 the fine-grained parallel search in line 4 cannot be fully supported in the current nested *forall* structure in Chapel. So, the high speedup for real-world graphs should be attributed to our optimization method that may exploit the skewed degrees in real-world graphs instead of fine-grained parallelism. The performance comparison between *MS* and *MS-2* directly shows how much we can gain by exploiting the skewed degrees in three adjacency lists in *K-Truss* analysis. For all the real-world graphs, *MS* is better than *MS-2*. At the same time, *MS-2* is better than *IP* even though both use only two given adjacency lists. The reason is that *MS-2* uses binary search in another large adjacency list, but *IP* will visit every element in the large adjacency list. For the synthetic graphs, the three methods, *IP*, *MS*, and *MS-2* are close to each other because the degree distributions of synthetic graphs are not as skewed as those of real-world graphs. The higher difference in degrees of different vertices a graph has or the higher percentage of unbalanced degrees a graph has, the more performance improvement can be achieved by our *MS* method compared with the optimized *IP*, *MS-2* and *Dir* methods.

Fig. 4 shows the speedup results of different *Max-Truss* analysis algorithms compared with *Naïve IP* as the baseline. We can see a similar trend just as in Fig. 3. Our *MS* method can achieve the best performance for all the real-world graphs. The highest speedup is 83.56 on graph "com-youtube", and the average speedup is 22.16. For the synthetic graphs, the performance of *IP*, *MS-2* and *MS* are very close to each other. The result shows that our optimized *Max-Truss* algorithm can use very small search steps to identify the $Max - Truss$ subgraph.

Fig. 5 gives the speedup results of different *Truss Decomposition* algorithms compared with the *Naïve IP* algorithm as the baseline. *Truss Decomposition* analysis is the most time-consuming method because it needs to find all subgraphs with different truss values. However, it can give much more detailed information that cannot be provided by *K-Truss* and *Max-Truss* algorithms. Our *MS* method can achieve the best performance for all the real-world graphs. The maximum speedup is 568.96 on graph "soc-Epinions1" whose maximum degree is 3044 and the average degree is 10. The average speedup is 129.35 for all the graphs. Most of the speedups achieved by our *MS* method in *Truss Decomposition* are much better than the speedups in the corresponding *K-Truss* and *Max-Truss* analysis. The results show that the more search cases are needed, the more optimization opportunities can be captured by our *MS* method to improve the performance.

The experimental results of *K-Truss*, *Max-Truss* and *Truss Decompostion* analysis show that our different *MS* kernel-based optimized methods can take advantage of the properties of real-world graphs to improve the performance of our truss analytics algorithms significantly.

We do the same experiments on Platform 2 with real-world graphs, and the trend is very similar to that on Platform 1. For *K-Truss* analysis, our *MS* method can achieve the highest 162.51 speedups on graph "com-youtube". The average speedup is 26.30. For *MAX-Truss* analysis, the highest speedup is 56.15 on the same graph. The average speedup is 12.93. For *Truss Decomposition* analysis, our *MS* method can achieve 503.54 speedups on graph "soc-Epinions1". The average speedup is 158.31 (we do not include the detailed results of every graph because of the limit in space). On platform 2, our *MS* method can achieve a similar performance improvement. However, the absolute execution time on platform 2 is much shorter than on platform 1 for the same graph.

In one recent research paper, Conte et al. [11] developed a highly optimized *List Intersection* based truss decomposition algorithm using C++. They use SSE-acceleration when the two adjacency lists have a similar size. If the sizes of the two adjacency lists are very different, they will use a binary search-like method to avoid searching on the complete adjacency list. They try to avoid atomic operations to improve parallel performance in the triangle counting part. Their code[6] is also publically available, and they use OpenMP for parallel execution. We compare our *MS* based truss decomposition method with their *List Intersection*-based truss decomposition method on the same real-world graphs. The results are given in Fig. 6. We can see that for all the real-world graphs, the performance of our *MS* based *Truss Decomposition* method is much better than their *List Intersection*-based *Truss Decomposition* method. The highest speedup is 385.8 on "ca-GrQc" graph and the average speedup for all the real-world graphs is 128.

VI. Related Work

GraphChallenge[7] is a vital community effort of academia and industry to develop new solutions for analyzing graphs and sparse data. *K-Truss* is one of the graph challenge algorithms. The seminal paper about truss decomposition is that by Cohen [10], who introduced the concept of *K-Truss*, motivating it as an effective community indicator.

Intersect path [17] is an efficient and typical algorithm to implement *List Intersection* in triangle counting. However, the major problem is that it only works on two adjacency lists and cannot leverage the third vertex's adjacency list to improve performance. Directed graph or directing edges-based methods [7], [19] are further performance optimizations, but they need to relabel the vertices based on their degrees to improve

[6]https://github.com/google-research/google-research/tree/master/truss_decomposition

[7]https://graphchallenge.mit.edu/challenges

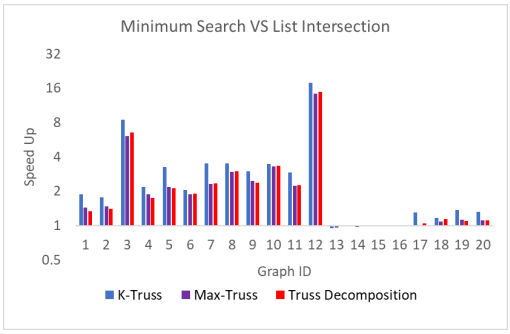

Fig. 2: Speedup of Minimum Search VS List Intersection.

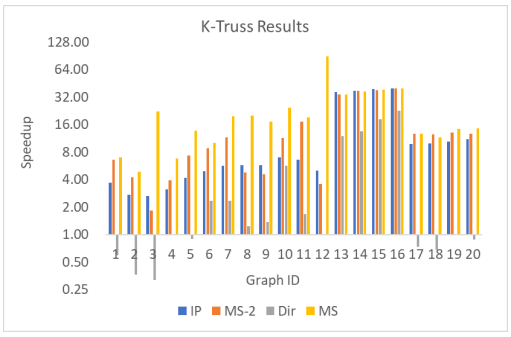

Fig. 3: Speedup of different K-Truss algorithms.

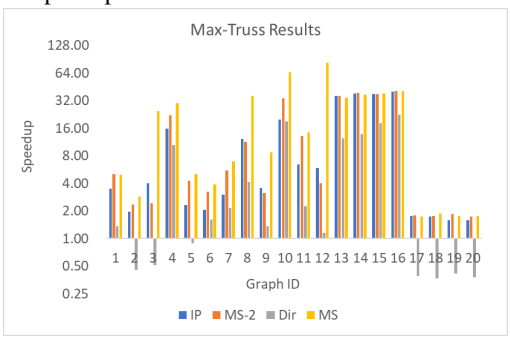

Fig. 4: Speedup of different Max-Truss algorithms.

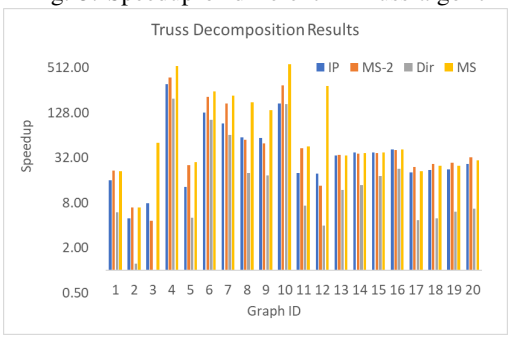

Fig. 5: Speedup of different Truss Decomposition algorithms.

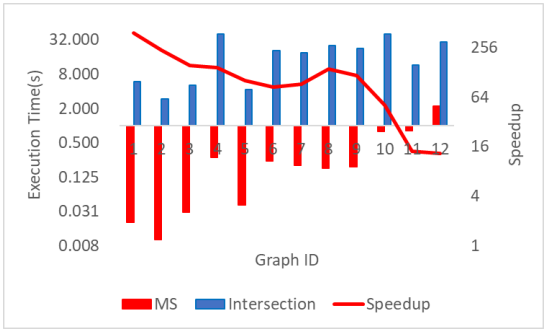

Fig. 6: Execution time of our MS algorithm and Intersection algorithm. The second y-axis is the speedup of our MS algorithm compared with the Intersection algorithm.

the counting performance. Such vertex-centric parallel methods are not suitable for an edge-centric parallel framework. Recently, Aberger et.al [1] exploits the SIMD instructions to improve the performance of set intersections. Han et. al [18] developed a binary presentation method together with the SIMD instructions to improve the performance of set intersection further. The above methods formulate triangle counting as a sequential problem and use SIMD to parallelize some operations. The proposed methods can improve the parallelism but do not reduce the total operations. However, our novel method formulates triangle counting as a parallel problem and reduces the total number of operations.

Besides the CPU platforms, many works for truss analysis are on GPUs or both CPUs and GPUs. Blanco et al. [8] presents a linear-algebraic formulation of the *K-Truss* graph algorithm and demonstrates the efficiency of their fine-grained parallel approach on both CPU and GPU. Almasri et al. [3] uses multiple GPUs to improve the binary search *Max K-Truss* performance. Date et al. [12], [21] takes advantage of the heterogeneous platform (CPU+GPU) to improve the performance. Most of the current works are on static graphs. Green et al. [16] uses a new dynamic graph formulation to achieve scalable performance on GPUs for both *K-Truss* and *Max K-Truss* analysis.

Diab et al. [13] explores the design space of different optimizations on GPUs including edge-centric [3], [12], [21], [27] and vertex-centric parallelization [6], directing edges by degree [7], [19], tiling [19], [29], parallelizing intersections [7], [19], removing deleted edges intermediately [7], [8], [12], and recomputing support values to achieve better performance for specific input graphs [3], [7], [12], [16], [21]. Integrating different optimization technologies together for different practical scenarios is a challenging work. Shi et.al [25] optimizes their truss decomposition algorithm by compacting intermediate results, dynamically adjusting the computation and parallelizing on both CPU and GPU together. Our novel *Minimum Search* method can work together with different kinds of high-level truss analysis methods.

VII. Conclusion

Interactive and productive *Truss* analytics is critical to exploit the value of large networks at scale. We have developed

a novel triangle counting kernel *Minimum Search* that can significantly reduce the total number of search operations by exploiting the power-law distribution property in real-world graphs. Instead of just working on two given adjacency lists like the widely adopted *List Intersection* method, the proposed *Minimum Search* method can completely avoid the largest adjacency list and select the smaller adjacency list to check if three vertices can form a triangle. Based on our novel kernel, highly optimized *Truss* analytics algorithms are developed. Experimental results show that the performance of our novel *Minimum Search* method is much better than that of the extensively used *List Intersection* method on different kinds of real-world graphs. Furthermore, all our truss analytics algorithms have been integrated into an open-source explore data analysis framework Arkouda to enable high-performance graph analytics through the easy-to-use Python interface. All the code is publicly available on GitHub [24].

ACKNOWLEDGMENT

We appreciate the help from the Arkouda and the Chapel community when we integrated the algorithms into Arkouda. This research was funded in part by NSF grant number CCF-2109988.

REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[2] Lada A Adamic, Bernardo A Huberman, AL Barabási, R Albert, H Jeong, and G Bianconi. Power-law distribution of the world wide web. *science*, 287(5461):2115–2115, 2000.

[3] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Update on k-truss decomposition on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[4] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph partitioning and graph clustering*, volume 588. American Mathematical Society Providence, RI, 2013.

[5] John T Behrens. Principles and procedures of exploratory data analysis. *Psychological Methods*, 2(2):131, 1997.

[6] Mauro Bisson and Massimiliano Fatica. Static graph challenge on GPU. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2017.

[7] Mauro Bisson and Massimiliano Fatica. Update on static graph challenge on GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2018.

[8] Mark Blanco, Tze Meng Low, and Kyungjoo Kim. Exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[9] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[10] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency technical report*, 16(3.1), 2008.

[11] Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. Truly scalable k-truss and max-truss algorithms for community detection in graphs. *IEEE Access*, 8:139096–139109, 2020.

[12] Ketan Date, Keven Feng, Rakesh Nagi, Jinjun Xiong, Nam Sung Kim, and Wen-Mei Hwu. Collaborative (CPU+GPU) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[13] Safaa Diab, Mhd Ghaith Olabi, and Izzat El Hajj. Ktrussexplorer: Exploring the design space of k-truss decomposition optimizations on GPUs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2020.

[14] Zhihui Du, Oliver Alvarado Rodriguez, Joseph Patchett, and David A Bader. Interactive graph stream analytics in arkouda. *Algorithms*, 14(8):221, 2021.

[15] Irving J Good. The philosophy of exploratory data analysis. *Philosophy of science*, 50(2):283–295, 1983.

[16] Oded Green, James Fox, Euna Kim, Federico Busato, Nicola Bombieri, Kartik Lakhotia, Shijie Zhou, Shreyas Singapura, Hanqing Zeng, Rajgopal Kannan, et al. Quickly finding a truss in a haystack. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[17] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2014.

[18] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602, 2018.

[19] Yang Hu, Pradeep Kumar, Guy Swope, and H Howie Huang. Trix: Triangle counting at extreme scale. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[20] Andrew T Jebb, Scott Parrigon, and Sang Eun Woo. Exploratory data analysis as a foundation of inductive research. *Human Resource Management Review*, 27(2):265–276, 2017.

[21] Vikram S Mailthody, Ketan Date, Zaid Qureshi, Carl Pearson, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. Collaborative (CPU+ GPU) algorithms for triangle counting and truss decomposition. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[22] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, pages 28–28, 2019.

[23] William Reus. CHIUW 2020 Keynote: Arkouda: Chapel-Powered, Interactive Supercomputing for Data Science. In *Chapel Implementers and Users Workshop, 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 650–650. IEEE, 2020.

[24] Oliver Alvardo Rodriguez, Zhihui Du, Joseph T. Patchett, Fuhuan Li, and David A. Bader. Arachne: An Arkouda package for large-scale graph analytics. In *The 26th Annual IEEE High Performance Extreme Computing Conference (HPEC), Virtual, September 19-23, 2022*, 2022.

[25] Jessica Shi, Laxman Dhulipala, and Julian Shun. Theoretically and practically efficient parallel nucleus decomposition. *arXiv preprint arXiv:2111.10980*, 2021.

[26] Andrew T Stephen and Olivier Toubia. Explaining the power-law degree distribution in a social commerce network. *Social Networks*, 31(4):262–270, 2009.

[27] Chad Voegele, Yi-Shan Lu, Sreepathi Pai, and Keshav Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[28] Jia Wang and James Cheng. Truss decomposition in massive networks. *arXiv preprint arXiv:1205.6693*, 2012.

[29] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan Berry, Michael Wolf, Jeffrey S Young, and Ümit V ÇatalyÜrek. Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments:(update on static graph challenge). In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–4. IEEE, 2019.