Arachne: An Arkouda Package for Large-Scale Graph Analytics

Oliver Alvarado Rodriguez, Zhihui Du, Joseph Patchett, Fuhuan Li, David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
{oaa9,zd4,jtp47,fl28,bader}@njit.edu

***Abstract*—Due to the emergence of massive real-world graphs, whose sizes may extend to terabytes, new tools must be developed to enable data scientists to handle such graphs efficiently. These graphs may include social networks, computer networks, and genomes. In this paper, we propose a novel graph package, Arachne, to make large-scale graph analytics more effortless and efficient based on the open-source Arkouda framework. Arkouda has been developed to allow users to perform massively parallel computations on distributed data with an interface similar to NumPy. In this package, we developed a fundamental sparse graph data structure and then built several useful graph algorithms around our data structure to form a basic algorithmic library. Benchmarks and tools were also developed to evaluate and demonstrate the use of our graph algorithms. The graph algorithms we have implemented thus far include breadth-first search (BFS), connected components (CC), k-Truss (KT), Jaccard coefficients (JC), triangle counting (TC), and triangle centrality (TCE). Their corresponding experimental results based on real-world and synthetic graphs are presented. Arachne is organized as an Arkouda extension package and is publicly available on GitHub (https://github.com/Bears-R-Us/arkouda-njit).**

***Index Terms*—graph analytics, large-scale data, parallel algorithm, open-source framework**

I. Introduction

Graphs are widely used to represent real-world problems in numerous domains, such as in the social sciences and in biological and information systems. However, the increasing scale of graphs prevents them from being analyzed by popular exploratory data analysis tools, such as Python. Arkouda [20], [21] is an open-source software framework created to be a replacement for NumPy at scale in Python. Arkouda is powered by Chapel [4] at the back-end with a front-end facing Python interface. Chapel is an open source high-level programming language designed for productive parallel computing at scale. It enables the usage of distributed data structures to easily spread data across many computing nodes and their memories. It further provides a global namespace to access all variables without knowing their physical memory locations. Finally, Chapel code is typically easier to understand with structures made specifically to facilitate data and task parallelism.

The major objective of this research is to enable data scientists to productively handle large-scale graphs that can only be held by powerful back-end servers. In other words, we want data scientists to be able to inspect large data as if they were handling it locally on their personal computing devices. There are many existing graph packages with similar objectives, such as the Knowledge Discovery Toolbox (KDT) [19] and GraphBLAS [16]. KDT also has a Python interface, but its back-end compute engine is Combinatorial BLAS. Compared with Combinatorial BLAS, Chapel can support productive and flexible graph algorithm development. GraphBLAS provides a linear algebra language for graph operations. However, data scientists may find graph analytics more productive in Python than through a linear algebraic interface.

This work aims to support large-scale graph analysis with high efficiency by developing the Arachne package. More knowledge and insights can be exploited from graph data easily and efficiently. Without Arachne, data scientists cannot handle real-world graphs productively because of scale, performance, and code complexity.

The major contributions of this research are as follows:

1) A fundamental sparse graph data structure at the core of Arachne is built in Chapel and integrated with Arkouda, accessible through a Python front end.
2) Arachne contains typical graph analytical algorithms to support productive graph analytics in Python, backed by Chapel. These graph algorithms are organized in a cohesive framework with all Chapel and Python code available on GitHub.
3) Experimental results on real-world and synthetic graphs that demonstrate the methods in Arachne.

II. Graph Infrastructure Design and Development

In this section, we introduce the *Double Index* (DI) graph data structure that is the backbone of Arachne. DI helps users conduct large-scale graph analytics productively and efficiently.

A. Graph Data Structure Design

Our graph infrastructure in Arachne focuses on large-scale, and oftentimes sparse, real-world graphs. The compressed sparse row (CSR) format is often and widely used to express sparse graphs. However, CSR is a vertex-centric data structure that cannot efficiently support edge-centric graph algorithms. Therefore, we designed and developed a novel *Double Index* data structure [10] that can efficiently support both vertex-centric and edge-centric graph algorithms, and serves as the core data structure in Arachne.

The graph data structure was developed into both the Python front end and Chapel back end. We defined a graph class at the Python front end to hold the graph metadata. The metadata includes: (1) the graph name ID, which is used to access the raw data at the back end; (2) the number of vertices and edges; (3) and flags to signify if the graph is directed and/or weighted. We also defined a graph query function *graph_query* to access the raw data of a given graph. We can query the edge arrays (source/destination of any edge), the vertex arrays (starting position in the edge arrays and the number of neighbors of any vertex), and the weight arrays (the weight of any edge).

We designed the corresponding graph class in Chapel at the back end to build the complete graph in memory. We defined an enumerated Chapel data type *Component* to keep names of different graph parts and build a map between the names and the corresponding data. The *Component* data type is very flexible in expressing different parts of a graph. The specific data organization can be changed in any way, but all graphs can share the same abstract class description. We only needed to develop three kinds of functions for each component: (1) Checking if the such component exists. (2) Setting to store the raw data in the component memory. (3) Querying to return the raw data stored in the component memory. We utilized the existing symbol table mechanism and developed a specific graph symbol table entry class *GraphSymEntry* to maintain graph objects. A one-to-one mapping between a graph name ID and a graph object is built in the symbol table, and the Python front end can query a graph component using a graph name ID based on the symbol table.

Besides the default Chapel data array distribution, we developed another efficient edge-vertex-locale mapping mechanism by providing a *DomArray* data structure, which can customize the array size based on a user-specified Chapel domain, to improve data access locality. The basic idea is shown in Fig. 1. The degree distributions of real-world graphs are often highly skewed. Therefore, edge-centric graph partitions may evenly distribute the edges onto different locales/computing resources. This method will help to achieve near-perfect load balance.

Furthermore, for all the edges assigned to a specific locale[1] (a locale is a computing resource in Chapel), we may map the corresponding vertices onto the same locale. This will cause an irregular vertex array distribution. However, it can improve the memory access locality when we employ some edge-centric graph algorithms. We assign the workload based on the edge distribution to improve load balancing and memory access locality for all implemented graph algorithms in the package.

B. The Arachne Graph Toolkit

We have developed basic graph operations to form a toolkit to support different applications of graph analytics. The *graph_file_read_mtx()* Python function is used to read an "mtx" format graph file into the back end. A graph must be built into memory before our toolkit can process it. *graph_file_read()* is a similar Python function with high flexibility. It can skip some head metadata parts to read the real graph data. At the Chapel back-end, we use distributed arrays to store the graph data when the Arkouda server is started with more than one compute node. When this happens, parallel IO will be employed, and each local array will only keep the data owned by itself. For streaming graphs, we provide the *stream_file_read()* to read a stream as a file and build the corresponding graph sketch in the back end.

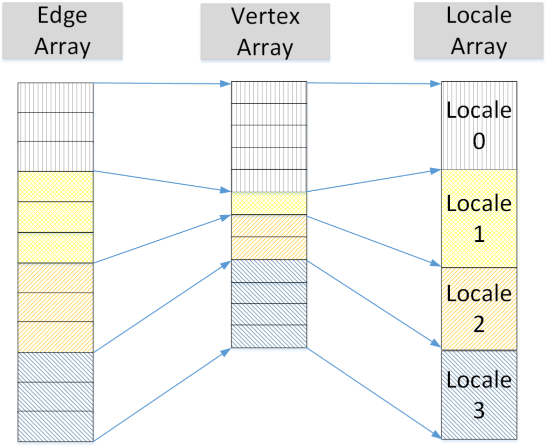


Fig. 1: Edge-centric sparse graph Edge-Vertex-Locale mapping mechanism to improve memory access locality.

Many graph algorithms do not allow duplicated edges or self-loops and the vertices usually must be labeled starting from 0. We developed the *graph_file_preprocessing()* function to clean the original graph. Besides removing the duplicated edges and self-loops, we also have the following operations. (1) If the *Remap* flag is set, we will map all the N vertices in the graph from 0 to $N-1$. (2) If the *DegreeSort* flag is set, we will relabel the vertex based on their degrees. (3) If the *RCM* flag is set, we will preprocess the graph using the *reverse Cuthill–McKee* algorithm. (4) If the *AlignedArray* flag is set, we will further map the vertex arrays employing the Edge-Vertex-Locale mechanism instead of the default distribution.

For benchmarking we also developed the *rmat_gen()* function to generate *R-MAT* graphs [3] based on user-given parameters. When we conduct graph analysis on *R-MAT* graphs, we may use the provided function to generate the required graphs, which all our integrated graph algorithms can then process.

III. Implementations of Typical Graph Algorithms

This section introduces useful graph algorithms we have developed and implemented in Arachne. The development of the algorithms take into account their implementation in Chapel. Therefore, the explanations of their parallelism are framed using Chapel's lexical and code structures. Furthermore, each algorithm has a corresponding benchmark file in the Arachne framework that shows how they can be utilized from the

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

Python front end. More information on the benchmarks is found in Section III-G.

A. Breadth First Search

Since *breadth first search (BFS)* is a very popular method in graph analysis, we developed our *BFS* implementation [9] in Chapel and integrated it into Arachne at the early stages of development.

The major feature of our *BFS* implementation in Chapel is taking advantage of the high-level data structure *DistributedBag*[2] to manage the current frontier and the next frontier. Vertices on different locales can be added into the *DistributedBag* in a parallel-safe manner. This makes parallel programming more efficient because we do not need to implement detailed array insertions or appends at a low level.

The *RCM* preprocessing method provided by our preprocessing function is used to improve the performance of *BFS*. When using a distributed computing environment, each locale will only work on the vertices whose edges are on that specific locale. Through this workload balance we can take advantage of the locality to extend the next frontier in parallel. The irregular memory access pattern of *BFS* makes its performance in a distributed computing environment often worse than that of a shared-memory *BFS* counterpart. We believe this is due to many remote fine-grained writes occurring during frontier expansion [14]. This can be mitigated by implementing aggregation when performing remote writes. However, this is out of the scope of this work and will be revisited in later updates to Arachne.

B. Connected Components

Building a connected components algorithm is rather easy based on our implemented *BFS* method. However, the *BFS* based method cannot achieve high performance for a large graph with a large diameter. Therefore, we implemented the *FastSV* algorithm [22] in Chapel to improve the performance of the connected components method. *FastSV* involves iterating over all the edges in a graph multiple times until convergence. This edge-centric computational model is attractive for our edge-based distribution data structure. Each locale can process its edges independently to achieve good parallelism.

Furthermore, we simplified the three hooking methods in the *FastSV* algorithm to reduce the total number of synchronizations. At the same time, we designed a simple counter data structure to check the convergence quickly instead of employing two arrays to compute their difference.

C. K-Truss Analytics

The elemental kernel in *K-Truss* algorithms is finding all the triangles in a graph. We developed a novel *Parallel Minimum Search* triangle search kernel for Arachne that can significantly reduce the total triangle search time for any algorithms that rely extensively on triangle searching. Compared with the widely used list intersection method, our *Parallel Minimum Search* method can save many operations, and its performance is better than that of both the *List Intersection* [7] method, and the directed graph-based method [1]. Based on this triangle search kernel, we have implemented *K-Truss*, *Max-Truss* and *Truss Decomposition* algorithms [8] for different kinds of truss analysis. Furthermore, we employed a binary search-like method in *Max-Truss* to reduce the total number of searches. We utilize the results of K truss to accelerate the analysis of *K+1* truss in *Truss Decomposition*.

D. Jaccard Coefficient

The *Jaccard Coefficient* [17] is a graph analytic method to expose the similarity between two vertices of a graph in terms of the overlap of their neighboring vertices. For a given vertex w, let $Adj(w)$ be the adjacency list of w. Then, $\forall u, v \in Adj(w)$, we can increase the number of vertices in common between u and v by 1. This "backwards" method is implemented in Arachne based on our edge-centric graph partitioning with high memory access locality. Currently, the memory is not optimized for a highly sparse graph because we will use $N \times N$ (N is the total number of vertices) memory to hold all of the coefficients. We are working on significantly reducing the total memory and improving the locality by using a hash-based method to discard most of the zero coefficients.

E. Triangle Counting

Arachne contains our implementations of the exact triangle counting algorithm for static graphs and approximate triangle counting algorithm for streaming graphs [10]. We built a regression model for real-world graphs whose degrees follow power-law distributions and synthetic graphs whose degrees follow a normal distribution. Employing different regression models for different degree distributions can improve the accuracy of our approximate algorithm. In the Arachne toolkit, based on their practical performance, we employ two triangle counting methods from all the methods implemented in the *triangle centrality* analysis. We chose to highlight the *Parallel Minimum Search*-based and *Path Merge*-based triangle counting algorithms because they are the top two fastest methods for most of our test graphs.

F. Triangle Centrality

Triangle centrality [2] is a method to find the most important vertices based on the concentration of triangles surrounding each vertex. Based on our triangle counting algorithm, when we find a new triangle, we not only update the number of triangles of each vertex but also mark the three edges as true. This lets us know that the vertices connected by a true edge are connected to some triangle.

In Arachne, we reformed the triangle centrality formula so that we only need to calculate two items: (1) the sum of the number of triangles of the given vertex's adjacency list and itself, and (2) the sum of the number of triangles of the given vertex's neighbors that are connected by a triangle. Through this, we can calculate the triangle centrality more efficiently. Of course, all the triangle counting performance optimization methods are also used in the triangle centrality algorithm implementation.

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

G. Benchmarks

We have corresponding benchmarks implemented in Python for all Arachne graph algorithms to evaluate their performance and correctness. These graph benchmarks return the elapsed time of the algorithm and check the correctness for different inputs. For example, to evaluate the *K-Truss* algorithms, we developed *truss-setk.py*, *truss-max.py*, and *truss-deco.py* to evaluate one input graph or a batch of many graphs for *K-Truss*, *Max-Truss*, and *Truss Decomposition*, respectively. After we call the *graph_file_read()* to build the graph in memory, we call the *graph_ktruss(Graph,k)* function to execute different truss analyses. If the parameter value $k > 0$, then *K-Truss* analysis algorithm will be called. If k==-1, then the *Max-Truss* analysis algorithm is used. If k==-2, then we will call the *Truss Decomposition* analysis algorithm at the back-end to get the truss values of all edges. The proposed benchmarks can also be utilized by a new user as a tutorial on how their Python scripts or Jupyter notebooks should look like if they are using Arachne and/or Arkouda for graph analysis.

IV. Experimental Results

In this section, we provide the performance results of Arachne's algorithms on real-world and synthetic graphs.

A. Data Sets

The graphs used for experiments can be found in Table I. Columns give the graph name, the number of edges, the number of vertices, the number of connected components in the graph, and the maximum value of K from *Max-Truss* analysis. A mix of real-world and synthetic graphs was utilized for the experiments. The real-world graphs represent typical networks that typical data scientists may handle. These real-world graphs consist of authorship, email, and social networks. The largest real-world graph we used for testing with "complete" (ignoring the unoptimized Jaccard coefficient) results is *friendster* with about two billion edges and sixty-five million vertices.

Real-world graphs often not only have a different number of edges but different graph properties, such as diameters and power-law exponents. We measured how those factors can affect the performance of different graph algorithms. If graphs have a similar pattern, we can see how the size of the graphs can affect the algorithm's performance. The Delaunay synthetic graphs will double their number of edges with the same topological structure. Therefore, Delaunay graphs are an ideal synthetic benchmark to evaluate the algorithm scalability when the number of edges increases.

B. Experimental Results

Experiments were conducted on a high-performance server with 2 x Intel Xeon E5-2650 v3 @ 2.30GHz CPUs with 10 cores per CPU and a RAM capacity of 512GB.

Results in Table II are an average of over ten trials per data set per algorithm. Except for the graph name, each column is the execution time of different operations or algorithms. We list the graph algorithms *BFS*, *Connected Components* (Conn-Comps), *Jaccard Coefficient* (Jaccard), *K-Truss, Max-Truss*, and *Truss Decomposition* (Truss-Dec). For *Triangle Counting*, we give the results of two methods, one is the minimum search-based method [8], and another is the merge-path-based method [11]. We use the two methods to measure the *Triangle Counting* (Tricnt-Min for minimum search-based method and Tricnt-Merge for merge-path-based method) and the corresponding *Triangle Centrality* (Trictr-Min for minimum search method and Trictr-Merge for merge-path-based method) algorithms.

TABLE I: Basic information for the graphs used for experimentation.

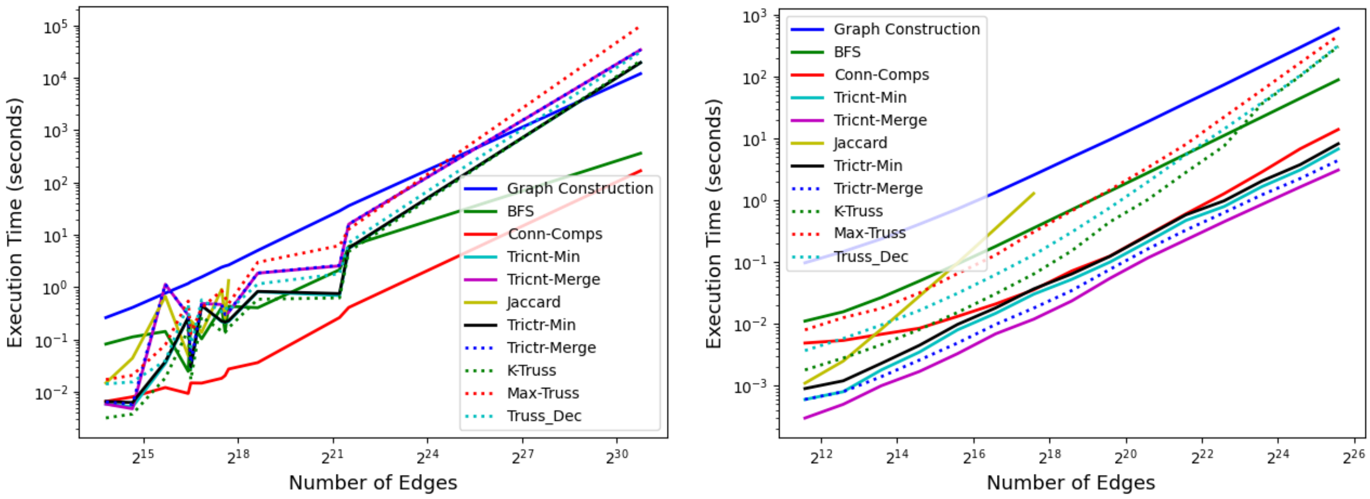
Graphs	Edges	Vertices	CCs	Triangles	Max K
ca-GrQc	14484	5242	354	48260	44
ca-HepTh	25973	9877	427	28339	32
as-caida20071105	53381	26475	1	36365	16
facebook_combined	88234	4039	1	1612010	97
ca-CondMat	93439	23133	567	173361	26
ca-HepPh	118489	12008	276	3358499	239
email-Enron	183831	36692	1065	727044	22
ca-AstroPh	198050	18772	289	1351441	57
loc-brightkite_edges	214078	58228	547	494728	43
soc-Epinions1	405740	75879	2	1624481	33
amazon0601	2443408	403394	7	3986507	11
com-Youtube	2987624	1134890	1	3056386	19
friendster	1806067135	65608366	1	4173724142	129
delaunayn10	3056	1024	1	2047	4
delaunayn11	6127	2048	1	4104	4
delaunayn12	12264	4096	1	8215	4
delaunayn13	24547	8192	1	16442	4
delaunayn14	49122	16384	1	32921	4
delaunayn15	98274	32768	1	65872	4
delaunayn16	196575	65536	1	131842	4
delaunayn17	393176	131072	1	263620	4
delaunayn18	786396	262144	1	527234	4
delaunayn19	1572823	524288	1	1054626	4
delaunayn20	3145686	1048576	1	2109090	4
delaunayn21	6291408	2097152	1	4218386	4
delaunayn22	12582869	4194304	1	8436672	4
delaunayn23	25165784	8388608	1	16873359	4
delaunayn24	50331601	16777216	1	33746670	4

1) Graph Construction Time: First, we can see in Table II that the time of building the graphs in Arachne is generally much larger than the execution time of different algorithms. However, once a graph is built in memory, the graph data can be shared by all the graph algorithms implemented into Arachne. Therefore, the cost of the graph construction can be amortized by all the following analyses and querying operations on the graph.

2) Execution time of Different Algorithms: For each graph, in general, the execution time of *Connected Components* is the shortest. However, the execution time of *Jaccard Coefficients* is often the longest. This is expected because our highly optimized *Connected Components* can work on all the edges in parallel to quickly propagate the root vertex IDs of all the components. The workload is balanced, and no idle threads exist in this procedure. It will need at most $log(\frac{M}{P})$ iterations on all the edges to converge, where M is the total number of edges and P is the total number of parallel processors or cores. However, calculating the *Jaccard Coefficients* will need $\frac{N \times N}{2}$ memory (if we keep all the coefficients) and about $(\frac{N}{P})^2 \times \frac{M}{P}$ calculations to get all the *Jaccard Coefficients*, where N is the

TABLE II: Execution time (seconds) of different graph algorithms on real-world and synthetic graphs (‘-’ means that the graph cannot be held by current memory).

Graphs	Graph Construction	BFS	Conn-Comps	Tricnt-Min	Tricnt-Merge	Jaccard	Trictr-Min	Trictr-Merge	K-Truss	Max-Truss	Truss-Dec
ca-GrQc	0.2641	0.0825	0.0066	0.0059	0.0059	0.0149	0.0066	0.0065	0.0032	0.0173	0.0143
ca-HepTh	0.4114	0.1128	0.0081	0.0052	0.0048	0.0443	0.0063	0.0058	0.0038	0.0209	0.0156
as-caida20071105	0.7743	0.1437	0.0122	0.0349	1.1407	0.6731	0.0381	1.1439	0.0186	0.0797	0.0397
facebook_combined	1.1833	0.0248	0.0094	0.2632	0.2954	0.0493	0.2645	0.2968	0.1539	0.5461	0.4720
ca-CondMat	1.2727	0.2101	0.0149	0.0279	0.0333	0.2309	0.0304	0.0358	0.0172	0.1311	0.0722
ca-HepPh	1.5751	0.1037	0.0148	0.4462	0.4898	0.1382	0.4489	0.4924	0.2748	0.4040	0.5768
email-Enron	2.4186	0.4124	0.0183	0.2207	0.4696	0.9165	0.2264	0.4746	0.1456	0.8673	0.3427
ca-AstroPh	2.5489	0.1431	0.0206	0.2122	0.2983	0.2599	0.2159	0.3021	0.1378	0.6402	0.4182
loc-brightkite_edges	2.6647	0.4336	0.0275	0.2213	0.3315	1.3466	0.2301	0.3400	0.1687	0.4883	0.3600
soc-Epinions1	5.0575	0.4074	0.0365	0.8237	1.8627	3.4277	0.8353	1.8745	0.5923	2.9999	1.1582
amazon0601	28.7435	2.1594	0.2645	0.6977	2.5684	–	0.7587	2.6287	0.6284	6.2614	1.8164
com-Youtube	36.0688	6.0387	0.4118	5.5000	16.0007	–	5.6162	16.1142	4.8273	12.8802	7.0787
friendster	12023.7931	362.7490	168.2291	19499.3154	34238.1467	–	19529.4456	34268.2398	21155.3487	98481.9	31356.5
delaunayn10	0.0973	0.0111	0.0049	0.0006	0.0003	0.0011	0.0009	0.0006	0.0018	0.0080	0.0037
delaunayn11	0.1476	0.0157	0.0054	0.0008	0.0005	0.0025	0.0012	0.0008	0.0028	0.0126	0.0058
delaunayn12	0.2365	0.0267	0.0068	0.0018	0.0010	0.0082	0.0023	0.0014	0.0046	0.0178	0.0096
delaunayn13	0.4092	0.0493	0.0085	0.0035	0.0017	0.0281	0.0045	0.0026	0.0081	0.0317	0.0165
delaunayn14	0.7222	0.0931	0.0132	0.0080	0.0033	0.0963	0.0098	0.0049	0.0153	0.0651	0.0312
delaunayn15	1.3312	0.1791	0.0210	0.0148	0.0069	0.3496	0.0181	0.0098	0.0306	0.1296	0.0625
delaunayn16	2.5501	0.3546	0.0355	0.0302	0.0119	1.2826	0.0367	0.0180	0.0640	0.3067	0.1306
delaunayn17	4.9490	0.7023	0.0717	0.0525	0.0235	5.0781	0.0642	0.0346	0.1518	0.6710	0.3046
delaunayn18	9.5101	1.3968	0.1244	0.0997	0.0544	–	0.1233	0.0760	0.4452	1.4655	0.7552
delaunayn19	18.6501	2.7768	0.2744	0.2140	0.1168	–	0.2681	0.1631	1.0228	3.3737	1.9792
delaunayn20	37.1980	5.5629	0.6024	0.4709	0.2289	–	0.5809	0.3286	2.8205	8.0215	5.4189
delaunayn21	73.5252	11.1037	1.2717	0.7893	0.4484	–	0.9756	0.6288	7.5073	22.0953	14.1159
delaunayn22	147.7980	22.1427	2.8730	1.6619	0.8573	–	2.0468	1.2239	36.4061	60.4223	37.8495
delaunayn23	295.7184	44.3626	6.7721	3.0825	1.6348	–	3.7533	2.2393	101.1284	166.3325	104.1138
delaunayn24	598.9867	88.3603	13.9053	6.7230	3.0720	–	8.1737	4.4086	306.1790	456.9431	315.1510



(a) Real-World Graphs

(b) Synthetic Graphs

Fig. 2: Execution time vs number of edges for real-world and synthetic graphs.

number of vertices. Triangle counting is the dominant part of triangle centrality calculations, so most of triangle centrality's time goes towards triangle counting. The performance of *BFS* is highly related to the graph's diameter. A larger diameter often means poor performance because limited parallelism can be exploited. For *Truss Analysis*, *K-Truss* analysis is part of *Truss Decomposition*. This means the execution time of *K-Truss* will be smaller than that of *Truss Decomposition*. We use estimated upper bound and binary search to find the maximum *K* value. Since the estimated upper bound is often larger than the exact maximum *K*, there will be some additional operations for *Max-Truss* analysis. However, for a very large maximum *K* value, the execution time of *Truss Decomposition* will be larger than that of *Max-Truss*. The experimental results show that when the estimated maximum *K* value is not very large, we may choose *Truss Decomposition* to get the maximum *K* value with better performance.

For the same algorithm on different real-world graphs, a larger number of edges is not necessarily indicative of a longer execution time. For example, *facebook_combined* has more edges than *as-caida20071105*. However, it has a shorter execution time for most graph algorithms (except the truss analysis algorithms). This is due to *facebook_combined* having much fewer vertices, and the access for different edges may have higher locality. Arachne can take advantage of such properties to improve performance. This example shows that not only the scale but also the topology of a given graph can greatly affect the performance of the same algorithm.

3) Algorithm Scalability: We use the Delaunay graphs to show the scalability of our algorithms on the same topological structure but for differently scaled graphs. We can see that when the number of edges (workload) is doubled, the execution times of our algorithms in Arachne are approximately also doubled for most of the cases. The execution time increases almost linearly when the workload increases. This means that no idle computing resources can be used to further improve the performance, or our algorithms have already fully taken advantage of the parallel resources. However, when the given graph becomes very large, the execution time will increase faster because memory becomes the major performance bottleneck instead of computing resources.

In Fig. 2, we give the relationship between execution time and the number of edges for both real-world graphs and synthetic graphs. We can clearly see that the execution time will increase linearly with the number of edges for the same topological structure in the Delaunay synthetic graphs. However, for real-world graphs, the relationship is often very irregular because the topology and number of edges can both affect the performance of a graph.

4) Most Edges Processed Per Second For Each Algorithm: To finalize the discussion of our results, we will now list the highest value of edges processed per second for each of our algorithms. The best edges/second performance is as follows. (1) **Graph Construction:** 150,208 edges per second were processed for the friendster graph. (2) **BFS:** 4,978,834 edges per second were processed for the friendster graph. (3) **Conn-Comps:** 11,107,035 edges per second were processed for the soc-Epinions1 graph. (4) **Tricnt-Min:** 8,164,161 edges per second were processed for the delaunayn23 graph. (5) **Tricnt-Merge:** 16,759,420 edges per second were processed for the delaunayn17 graph. (6) **Jaccard:** 2,829,629 edges per second were processed for the delaunayn10 graph. (7) **Trictr-Min:** 6,704,921 edges per second were processed for the delaunayn23 graph. (8) **Trictr-Merge:** 11,416,582 edges per second were processed for the delaunayn24 graph. (9) **K-Truss:** 6,889,389 edges per second were processed for the ca-HepTh graph. (10) **Max-Truss:** 1,242,132 edges per second were processed for the ca-HepTh graph. (11) **Truss-Dec:** 1,663,869 edges per second were processed for the ca-HepTh graph.

V. Related Work

One of the earliest works that mimic a productive front end with a highly parallel and compute-intensive back end is Matlab*P (2.0) [5], [6]. It aimed to provide a user-friendly interface for massive computations in the back end in an embarrassingly parallel manner. Another approach that aims to provide similar graph functionality is the Knowledge Discovery Toolbox (KDT), which uses a combinatorial BLAS back end with MPI that communicates with a Python front-end [19]. Our Arachne toolbox provides a few more options than theirs, such as the ability to perform truss analytics and compute the Jaccard coefficient of graphs.

GraphBLAS [16] is an application programming interface (API) to express graph algorithms in the language of linear algebra. It can be supported by different packages such as SuitSparse [18]. Compared to our toolbox, we do not require a data scientist to be aware of underlying linear algebra or mathematical methods; all they need to know is how they can use graph algorithms for their data analyses.

There are also graph packages in Chapel, such as the Chapel Graph Library (CGL) [15] and the Chapel Hypergraph Library (CHGL) [13]. In these packages, graphs are represented as hypergraphs, allowing constant time conversion between them. They use both the regular graph and hypergraph representations to drive their graph analytical algorithms. Our work in Arachne differs by focusing on graph analytics and not requiring some conversions to hypergraphs to fully carry out graph algorithmic steps.

Of course, there are also graph toolboxes that are coded solely in Python, such as NetworkX [12]. However, such packages cannot handle very large graphs and the performance is limited.

VI. Conclusion

A highly productive graph package and environment are essential for data scientists to efficiently exploit the value from big graph data. Thus, we present the Arachne toolkit for interactive graph analytics at scale. On one hand, we hope data scientists can use simple tools such as general programming in Python. On the other hand, we hope this package can provide very high performance for processing large graphs. We take advantage of the open-source framework Arkouda to build Arachne and integrate it into the Arkouda ecosystem to achieve the above two essential and often conflicting objectives of simplicity and large-scale analysis.

For future work with Arachne, we are going to integrate more graph algorithms and improve their performance through Chapel-based code and general algorithmic optimizations. Further, this paper highlights the performance of our algorithms in a shared-memory computational environment. However, to truly handle massive file sizes, we need to further explore the optimization methods of these algorithms in a distributed (memory) computing environment.

Acknowledgment

We thank the Chapel and Arkouda communities for their support as well as the funding support of the NSF through grant CCF-2109988.

References

[1] 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.

[2] Paul Burkhardt. Triangle centrality. *arXiv preprint arXiv:2105.00110*, 2021.

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

[4] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. Chapel comes of age: Making scalable programming productive. *Cray User Group*, 2018.

[5] Long Yin Choy. *MATLAB* P 2.0: Interactive supercomputing made practical*. PhD thesis, Massachusetts Institute of Technology, 2002.

[6] R. Choy and A. Edelman. Parallel MATLAB: Doing it Right. *Proceedings of the IEEE*, 93(2):331–341, 2005.

[7] 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.

[8] Zhihui Du, Joseph Patchett, Oliver Alvarado Rodriguez, and David A Bader. Truss analytics algorithms and integration in Arkouda. In *9th Annual Chapel Implementers and Users Workshop (CHIUW 2022)*, 2022.

[9] Zhihui Du, Oliver Alvarado Rodriguez, David A Bader, Michael Merrill, and William Reus. Exploratory large scale graph analytics in Arkouda. In *8th Annual Chapel Implementers and Users Workshop (CHIUW 2021)*, 2021.

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

[11] Oded Green, Saher Odeh, and Yitzhak Birk. Merge path-a visually intuitive approach to parallel merging. *arXiv preprint arXiv:1406.2628*, 2014.

[12] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[13] Louis Jenkins, Tanveer Bhuiyan, Sarah Harun, Christopher Lightsey, David Mentgen, Sinan Aksoy, Timothy Stavcnger, Marcin Zalewski, Hugh Medal, and Cliff Joslyn. Chapel HyperGraph Library (CHGL). In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[14] Louis Jenkins, Jesun Sahariar Firoz, Marcin Zalewski, Cliff Joslyn, and Mark Raugas. Graph algorithms in PGAS: Chapel and UPC++. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2019.

[15] Louis Jenkins and Marcin Zalewski. Chapel Graph Library (CGL). In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, CHIUW 2019, page 29–30, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[17] Peter M Kogge. Jaccard coefficients as a potential graph benchmark. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 921–928. IEEE, 2016.

[18] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[19] Adam Lugowski, David Alber, Aydm Buluç, John R Gilbert, Steve Reinhardt, Yun Teng, and Andrew Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 930–941. SIAM, 2012.

[20] 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.

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

[22] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. FastSV: A distributed-memory connected component algorithm with fast convergence. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 46–57. SIAM, 2020.