Arachne: An Open-Source Framework for Interactive Massive-Scale Graph Analytics

Presenter: Oliver Alvarado Rodriguez
Advisor: David A. Bader
New Jersey Institute of Technology

CCF-2109988

Introduction

In this poster we introduce Arachne, a framework that enhances productivity in massive-scale graph analytics. Arachne offers kernels for efficient graph analysis such as connected components, breadth-first search, triangle counting, k-truss, amongst others. The kernels are integrated into a backend server written in Chapel and can be accessed through a Python application programming interface (API). Arachne delivers high performance for graph analysis, and we have assessed its capabilities with the Friendster social network that is comprised of 1,806,067,135 edges and 65,608,366 vertices. Arachne's backend server is compatible with Linux supercomputers, is easy to set up, and can be utilized through either Python scripts or Jupyter notebooks, which makes it a desirable tool for Python data scientists who have access to highly performing Linux compute clusters. Find Arachne on GitHub at https://github.com/Bears-R-Us/arkouda-njit.

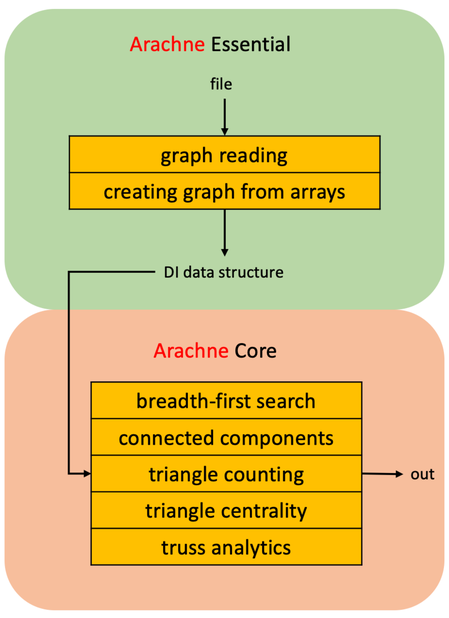

Figure 1: Arachne infrastructure.

Background

Chapel is a language that increases productivity for projects that require the partitioned global address space model (PGAS). PGAS allows a program to have a global view of memory across all the computational nodes that make up a system. There are still remote and local accesses, but access to remote data does not require elaborate code to handle it such as in programs written in C or C++ with OpenMP and MPI. Rather, the parallel structures in Chapel, such as *forall* loops, allow for computations to be performed across multiple nodes as if in a shared-memory system.

Motivation

We were tasked with providing graph kernels to run analysis on graph data stored in Arkouda's parallel and distributed arrays. The graphs could grow larger than a million edges and were to also store node labels, edge relationships, and properties. Property graphs were to be filtered to return simple graphs that could then be processed further by our graph kernels. Lastly, the API should match NetworkX's.

Contributions

1. A flexible double-index (DI) data structure for simple undirected graphs, directed graphs, and property graphs. Distributes graph data across compute nodes in an edge-centric and PGAS-friendly manner.
2. High-performing graph kernels for breadth-first search, connected components, truss analytics, triangle counting, and triangle centrality.
3. A novel minimum-search triangle counting algorithm and kernel.

Double-Index Data Structure

1. Given an edge index, we can extract all the vertex information in constant time including neighbors and any data held by that node.
2. Generating the neighborhood of a vertex takes time proportional with the size of its neighborhood.
3. The edges are equally distributed amongst locales where each locale can locally process its own edges.

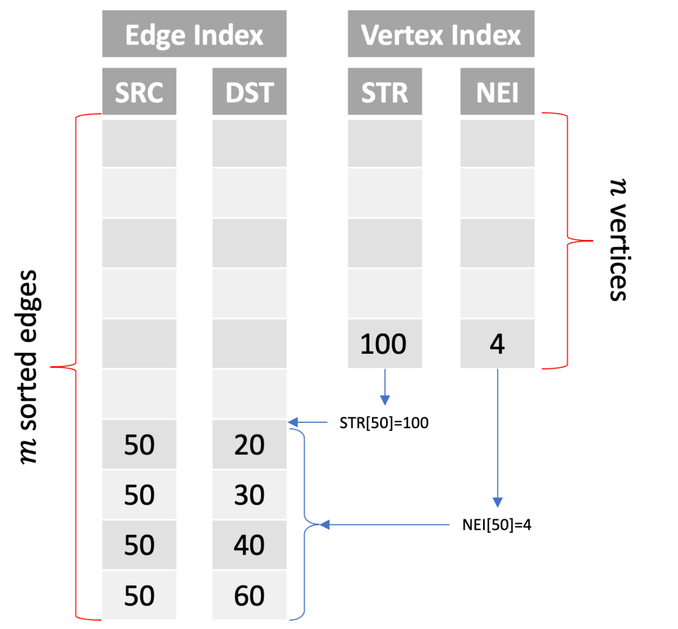

Figure 2: Double-index (DI) data structure.

Property Graphs in DI

1. Given an edge or vertex index, all the properties, labels, and relationships can be extracted in time proportional to the size of the storing list.
2. Generating simple subgraphs involve basic boolean indexing on Arkouda dataframes and calling the base graph generator provided by Arachne.

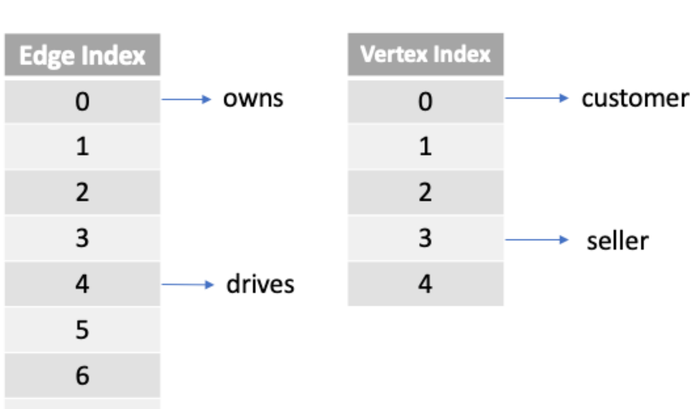

Figure 3: Property graphs in DI.

Graph Filtering

Property graphs can be generated from Arkouda dataframes, which are a collection of Arkouda arrays. Filtering these dataframes returns arrays that contain the nodes and edges that make up our subgraph. Then, a graph can be easily created by calling our `subgraph_view` method with the edges that make up our filter. A code snippet is shown below where "src" and "dst" make up the edges of the subgraph.

```
A = ak.arange(0, len(df), 1)
idx = df["col"] == "drives"
filter = df[idx]["src", "dst"]
subgraph = ar.subgraph_view(
    graph,
    ar.Graph(),
    filter_relationships=filter
)
```

Results

Kernel	Time (s)	TEPs
BFS	362.75	4978834
CC	168.23	10735759
KT_5	21155.35	85372
KT_M	31248.61	57797
KT_D	31356.50	57598
TRICNT	13339.06	135397
TRICTR	19590.32	92192

Table 1: Results on Friendster graph.

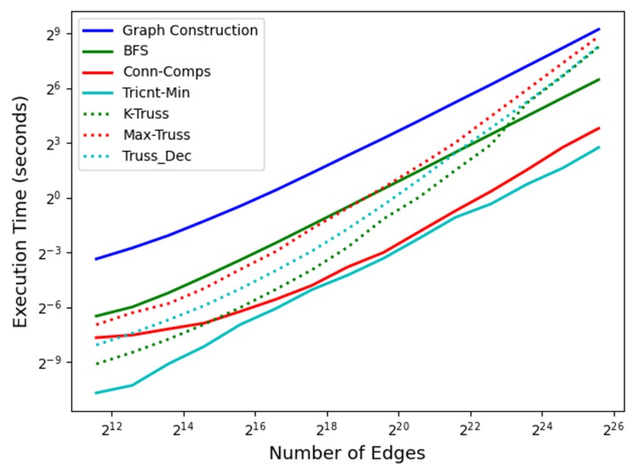

Figure 4: Scalability on synthetic graphs.

Conclusion

Most Python solutions for graph analysis only provide kernels for either sequential or shared-memory parallel systems. Arachne is built with the intention to be run on both shared-memory and distributed-memory systems. Future work includes optimizations targeting communication costs and implementing more kernels such as community detection. This is joint work with Zhihui Du, Joseph Patchett, Naren Khatwani, and Fuhuan Li.

References

1. Oliver Alvarado Rodriguez, Zhihui Du, Joseph Patchett, Fuhuan Li, David Bader (2022). *Arachne: An Arkouda Package for Large-Scale Graph Analytics*. IEEE HPEC.
2. Zhihui Du, Joseph Patchett, Oliver Alvarado Rodriguez, Fuhuan Li, David Bader (2022). *High-Performance Truss Analysis in Arkouda*. IEEE HiPC.