On the Design of Graph Analytical Software in Chapel

Oliver Alvarado Rodriguez, Zhihui Du, David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA

Supported by NSF grant: CCF-2109988
Introduction

• Graph analytical software consists of two main objectives: designing efficient **graph data structures** for fast data access and **algorithms** that exploit these efficient data accesses.
  • We have implemented an edge-based data structure based on a modified version of CSR we call the **Double-Index (DI)** data structure.
  • We have implemented algorithms for different graph analytical kernels such as **breadth-first search (BFS)**, triangle counting, connected components, etc.
  • All our functionality is bundled into the framework, Arachne, built on top of Arkouda.

• Firstly, this talk will present **DI** with a focus on new functionality to facilitate in-memory property graph analysis. Secondly, I will share our journey of optimizing **BFS** for distributed-memory execution in Chapel.
A Bird’s Eye-View of Arachne+Arkouda

Load in CSVs, HDF5s, Parquets, etc.

Generate or load graphs in from various sources.

Convert tabular data to a property graph with all data closely maintained with vertex and edges.

Perform analysis or filter for NetworkX, iGraph, or graph-tool.

1. import arkouda as ak
2. import arachne as ar
3. # Get src and dst from input file.
4. graph = ar.PropGraph()
5. # Generate label_df and relationships_df from input file.
6. graph.load_edge_attributes(relationships_df)
7. graph.load_node_attributes(label_df)
8. # User generates labels_to_find and relationships_to_find.
9. returned_nodes = graph.node_attributes["column"] == 1
10. returned_edges = graph.edge_attributes["column"] == 2
11. subgraph_src = ak.in1d(returned_edges[0], returned_nodes)
12. subgraph_dst = ak.in1d(returned_edges[1], returned_nodes)
13. kept_edges = subgraph_src & subgraph_dst
14. subgraph_src = subgraph_src[kept_edges]
15. subgraph_dst = subgraph_dst[kept_edges]
16. subgraph = ar.Graph()
17. subgraph.add_edges_from(subgraph_src, subgraph_dst)
18. # Run some other operations on subgraph! Reference our HPEC22 paper ☺

Easily usable through NetworkX-like API.

User edits a Python script or a Jupyter Notebook.

OPEN SOURCE: https://github.com/Bears-R-Us/arkouda-njit

PUBLICATIONS & PRESENTATIONS AT: HPEC, HiPC, IPDPS, & PPoPP

Chapel Server

ZMQ

Runs on any hardware, data stays in the back-end, user calls API through Python: powerful and productive. Server can run on supercomputers; Python API usable locally.

Original image source: https://chapel-lang.org/tiki/tiki/1158.png was modified for this presentation
Modular View of Arachne Functionality

Graph Construction

- add_edges_from()
- load_edge_attributes()
- load_node_attributes()
- read_matrix_market_file()
- rmat()
- gnp()
- random_tree()
- ...more!

Graph Algorithms

- bfs_layers()
- connected_components()
- triangles()
- triangle_centrality()
- squares()
- subgraph_isomorphism()
- diameter()

Data DI

- graph
- pdarray
- numeric

07 June 2024

Oliver Alvarado Rodriguez
Double-Index (DI) Data Structure

Examples and Persistence
(Property) **Graph Data Structure**

- Allows for simple, compact, distributable storage of vertex and edge sets.
- Given an edge index, $e$, all vertices that make up that edge are found in constant time, avoiding a binary search into SRC (CSR offsets index equivalent).
- MAP allows explicitly storing original vertex labels, returning original graph involves index operations SRC[MAP] and DST[MAP].
(Property) **Graph Data Structure**

<table>
<thead>
<tr>
<th>L1</th>
<th>IND</th>
<th>MAP</th>
<th>SEG</th>
<th>INT</th>
<th>UINT</th>
<th>REAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>34</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>69</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>89</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2</th>
<th>IND</th>
<th>SRC</th>
<th>DST</th>
<th>INT</th>
<th>UINT</th>
<th>BOOL</th>
<th>INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>-2</td>
<td>2</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>F</td>
<td>2</td>
</tr>
</tbody>
</table>

- Same **distributable** storage of vertex and edge attributes as base DI.
- Given an edge or vertex index, all attribute data can be easily accessed.
- Same storage principles apply to strings, which are stored in an object containing a byte array for characters and segments for where each string starts in the byte array.
- Sparse attribute arrays maintaining locality can also be created to only store attribute values that belong to a subset of indices.
Persisting Graphs via Arkouda Symbol Table

• Graph is stored as a GraphSymEntry which is a wrapper to SegGraph that inherits from CompositeSymEntry.

• Sparse arrays are stored in a SparseSymEntry (shoutout to Vass from the Chapel team) that inherits from GenSymEntry.

• We have other special classes to persist data such as maps, replicated arrays, and associative arrays. Plans to store “sparse” Arkouda categoricals and strings.
Breadth-First Search (BFS)

A Journey of Optimizations
General Information

• Important algorithm for solving problems that requires a complete traversal of a graph: answer questions like “how far is every other vertex from our source?”

• One of the fundamental graph algorithms in computer science.

• Has a sequential complexity of $O(n + m)$ where $n$ is the number of vertices and $m$ is the number of edges.
Single Locale Parallel BFS (version 1.0)

Input:

Output:

source vertex

Output: D = [0, 4, 1, 2, 3, 1, 3, 1, 2, 4]
Multilocal Parallel BFS (version 1.5)

Assume our edge list is split down the middle, then the neighborhood of some vertices will live on one compute node while the rest live on another compute node.

Input:

Output:

Output: D = [0, 4, 1, 2, 3, 1, 3, 1, 2, 4]

Any cross-color expansions are writes across the network; fine-grained writes hold up execution, large coarse-grained writes are better.
Each frontier is a list. Before we expand the frontiers in the following iteration, we aggregate them, and then write them to the appropriate frontier list.
Multilocal Parallel BFS Version 1.0

- Uses ideas of forward and reversed edges for undirected graphs. For example, $u \rightarrow v$ is stored in SRC and DST and $v \rightarrow u$ is stored in SRCr and DSTr.
- Use the “old” distributed bag to expand frontiers.
Multilocale Parallel BFS Version 1.5

- Combines the forward and reversed arrays to ensure every vertex has full access to its neighbors instead of a split view.
Multilocal Parallel BFS Version 2.0

Locale parallelism

Parallelism per-locale

Maintaining locality

Neighborhood expansion with aggregation

while true {
    var pending_work:bool = false;
    coforall loc in locales with {loc.reduce_pending_work, depth, frontier_sets} { 
        on loc { 
            var src_low = loc.localSubdomain().low;
            var src_high = loc.localSubdomain().high;
            forall v in frontier_sets[frontier_sets_idx] with {loc.reduce_pending_work} { 
                var adj_list_start = seg[i];
                var num_neighbors = seg[i+1] - adj_list_start;
                if (num_neighbors > 0) { 
                    var adj_list_end = adj_list_start + num_neighbors - 1;
                    
                    // Only pull the part of the adjacency list that is local.
                    var actual_start = max(adj_list_start, src_low);
                    var actual_end = min(src_high, adj_list_end);
                    
                    ref neighborhood = dist.localSlice(actual_start..actual_end);
                    for v in neighborhood { 
                        if (depth[v] == -1) { 
                            pending_work = true;
                            depth[v] = cur_level + 1;
                            var locs = find_locs(v, ranges);
                            for loc in locs do frontier_agg.copy(loc, v);
                        }
                    } 
                    // End forall
                    frontier_sets[frontier_sets_idx].clear();
                    // End on loc
                } 
            } 
        } 
    } 
    if (pending_work) { 
        cur_level = 1;
        frontier_sets_idx = (frontier_sets_idx + 1) % 2;
    } 
    // End while
    return "success";
## Multilocale BFS Communication Volume Heatmap

<table>
<thead>
<tr>
<th>locale</th>
<th>get 1.0</th>
<th>get 1.5</th>
<th>get 2.0</th>
<th>put 1.0</th>
<th>put 1.5</th>
<th>put 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15672640</td>
<td>7873842</td>
<td>639827</td>
<td>5629422</td>
<td>2749193</td>
<td>138070</td>
</tr>
<tr>
<td>1</td>
<td>15834332</td>
<td>7939017</td>
<td>687156</td>
<td>1952226</td>
<td>1016946</td>
<td>127936</td>
</tr>
<tr>
<td>2</td>
<td>15715554</td>
<td>7722659</td>
<td>226754</td>
<td>1942839</td>
<td>962031</td>
<td>45217</td>
</tr>
<tr>
<td>3</td>
<td>15817879</td>
<td>7723971</td>
<td>226880</td>
<td>1951313</td>
<td>962201</td>
<td>45060</td>
</tr>
<tr>
<td>4</td>
<td>15964559</td>
<td>7724880</td>
<td>226691</td>
<td>1961552</td>
<td>962199</td>
<td>51217</td>
</tr>
<tr>
<td>5</td>
<td>15739226</td>
<td>7726504</td>
<td>230024</td>
<td>1940688</td>
<td>962439</td>
<td>52714</td>
</tr>
<tr>
<td>6</td>
<td>15569450</td>
<td>7727678</td>
<td>229096</td>
<td>1925536</td>
<td>962680</td>
<td>51977</td>
</tr>
<tr>
<td>7</td>
<td>15341933</td>
<td>7736094</td>
<td>225083</td>
<td>1904757</td>
<td>963418</td>
<td>48413</td>
</tr>
</tbody>
</table>

1.0: 84 seconds (HPEC 21’)
2.0: 3.36 seconds

delaunayn20 is a graph with 3 million edges and a large diameter

**Takeaway:** Aggregating writes drastically reduces communication volumes, improving performance, all done easily through Chapel by adapting aggregators for different uses.
2.0 BFS Scalability

Grouped Execution Times per RMAT Scale

<table>
<thead>
<tr>
<th>RMAT Scale</th>
<th>2L</th>
<th>4L</th>
<th>8L</th>
<th>16L</th>
<th>32L</th>
<th>64L</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>2.11</td>
<td>3.43</td>
<td>5.87</td>
<td>8.10</td>
<td>9.66</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>2.14</td>
<td>3.69</td>
<td>6.35</td>
<td>10.28</td>
<td>13.04</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2.20</td>
<td>3.84</td>
<td>6.41</td>
<td>10.60</td>
<td>15.90</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1.93</td>
<td>3.09</td>
<td>6.84</td>
<td>9.86</td>
<td>15.56</td>
<td></td>
</tr>
</tbody>
</table>

Takeaway: As the number of locales increased, we see a good speed-up for distributed-memory breadth-first search.
Lessons Learned

• Using Chapel (or any PGAS-based languages and frameworks) don’t magically get rid of the complications of parallelizing and distributing graph operations.

• Adapting communication-aware optimizations, such as being aware of how neighborhoods are split across locales, can help improve graph-based performances.
Conclusion

• Using a programming language like Chapel allows us to quickly implement both shared-memory and distributed-memory algorithms to enable highly productive large-scale graph analysis.

• Using an existing framework like Arkouda allows us to focus more on graph algorithms while offloading tasks such as object persistence and array sorting.
Future Work

• Not everything needs to be distributed – large queries can be done in a distributed manner and smaller graphs analyzed on one compute node; can we *hybridize* our graph tools?

• Performance, performance, performance. Array-based operations are wonderful in Chapel, but do we need to build harnesses in Arachne to *call out to external programs* written in MPI, YGM, or other massively distributed tools?

• How can we *dynamically optimize* during runtime? For example, code regions that perform a lot of reads or writes on GASNet+Infiniband suffer when multiple parallel threads are writing since those values are transmitted sequentially. Chapel currently doesn’t allow for forall loops to dynamically use a runtime-given thread count.

• *There isn’t one data structure to rule them all.* Add capabilities in Arachne to build at runtime the data structure that is best for a given problem.
Thank You 😊
Questions?