On the Design of Graph Analytical Software in Chapel

<u>Oliver Alvarado Rodriguez</u>, Zhihui Du, David A. Bader

Department of Data Science

New Jersey Institute of Technology

Newark, NJ, USA

Supported by NSF grant: CCF-2109988

NJIT
New Jersey Institute of Technology

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 **D**ouble-**I**ndex (**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.

NJIT New Jersey Institute of Technology

A Bird's Eye-View of Arachne+Arkouda

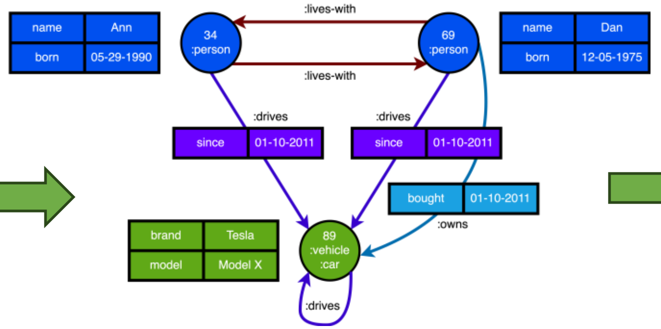
id	label	name	born	brand	model
34	person	Ann	1990	NULL	NULL
69					
89					
89					

src id	dst id	relationship	since	bought
34	69	lives-with	NULL	NULL
69	34	lives-with	NULL	NULL
34	89	drives	2011	NULL
69	89	drives	2011	NULL
69	89	owns	NULL	2011
89	89	drives	NULL	NULL

Load in CSVs, HDF5s, Parquets, etc.

- read_matrix_market_file()
- read_hdf()
- read_parquet()
- rmat()

Generate or load graphs in from various sources.

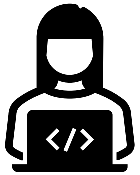


Convert tabular data to a property graph with all data closely maintained with vertex and edges.

- bfs_layers()
- subgraph_isomorphism()
- square_counting()
- subgraph_view()

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

User edits a Python script or a Jupyter Notebook.

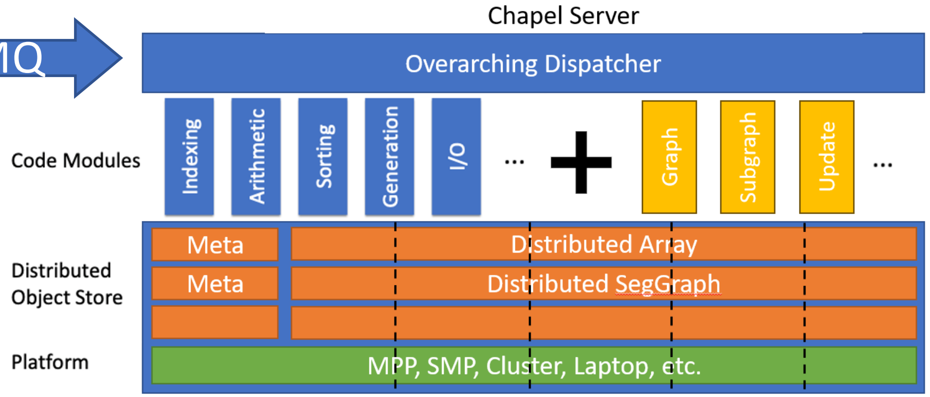


User

```python
import arkouda as ak
import arachne as ar

## Get src and dst from input file.

graph = ar.PropGraph()

## Generate label_df and relationships_df from input file.

graph.load_edge_attributes(relationships_df)
graph.load_node_attributes(label_df)

## User generates labels_to_find and relationships_to_find.
returned_nodes = graph.node_attributes[“column”] == 1
returned_edges = graph.edge_attributes[“column”] == 2

subgraph_src = ak.in1d(returned_edges[0], returned_nodes)
subgraph_dst = ak.in1d(returned_edges[1], returned_nodes)

kept_edges = subgraph_src & subgraph_dst

subgraph_src = subgraph_src[kept_edges]
subgraph_dst = subgraph_dst[kept_edges]

subgraph = ar.Graph()
subgraph.add_edges_from(subgraph_src, subgraph_dst)
## Run some other operations on subgraph! Reference our HPEC22 paper ☺
```

Easily usable through NetworkX-like API.

ZMQ



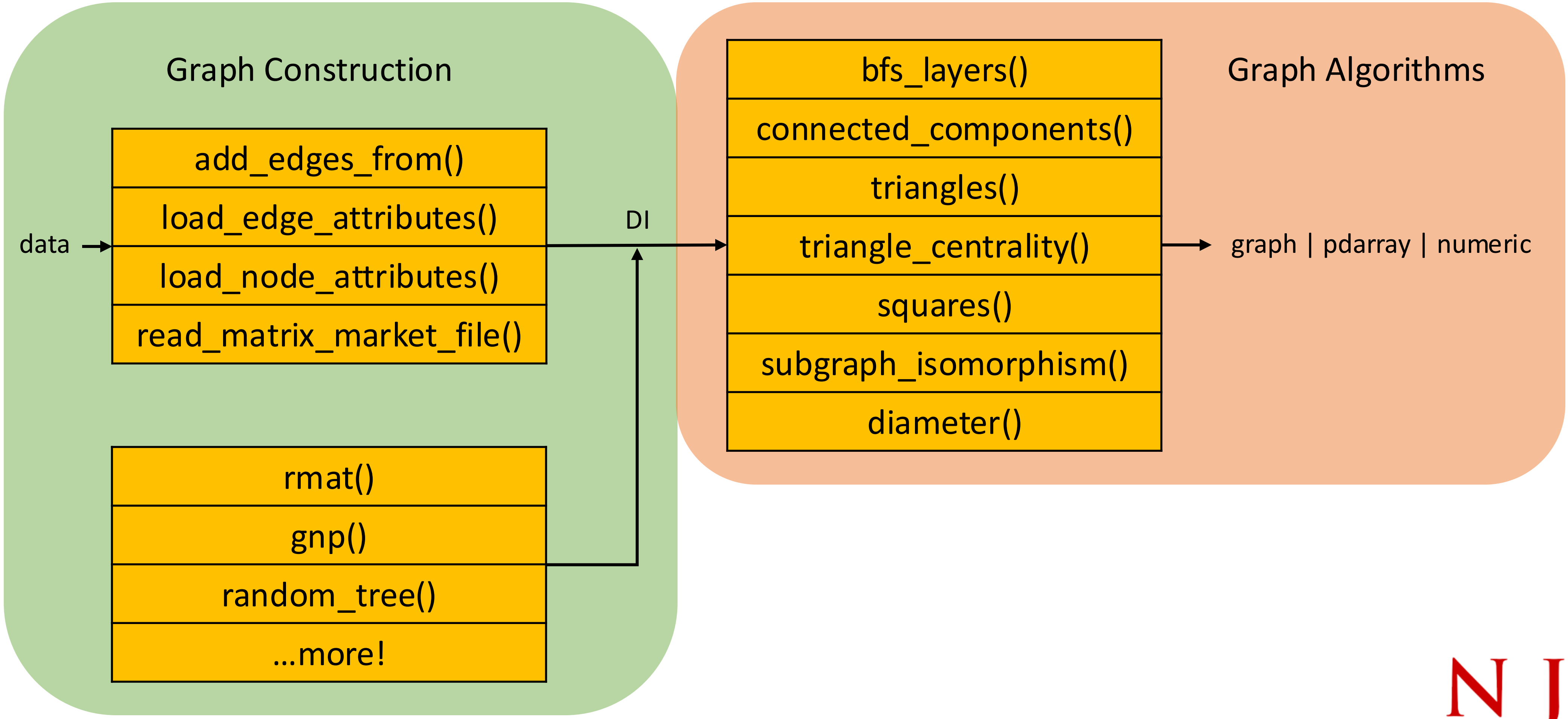
Original image source: https://chapel-lang.org/CHIUW/2020/Reus.pdf was modified for this presentation

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.

OPEN SOURCE: https://github.com/Bears-R-Us/arkouda-njit
PUBLICATIONS & PRESENTATIONS AT: HPEC, HiPC, IPDPS, & PPoPP

NJIT New Jersey Institute of Technology

Modular View of Arachne Functionality



Double-Index (DI) Data Structure

Examples and Persistence

(Property) **Graph** Data Structure

Vertices

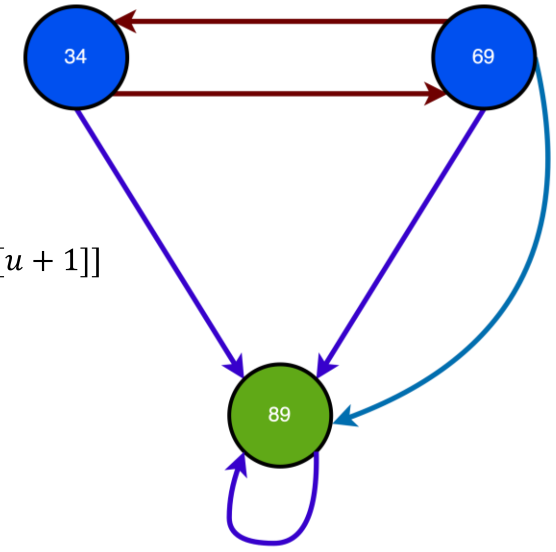
	IND	MAP	SEG
L1	0	34	0
	1	69	2
L2	2	89	4
	3		5

Edges

IND	SRC	DST
0	0	1
1	0	2
2	1	0
3	1	2
4	2	2

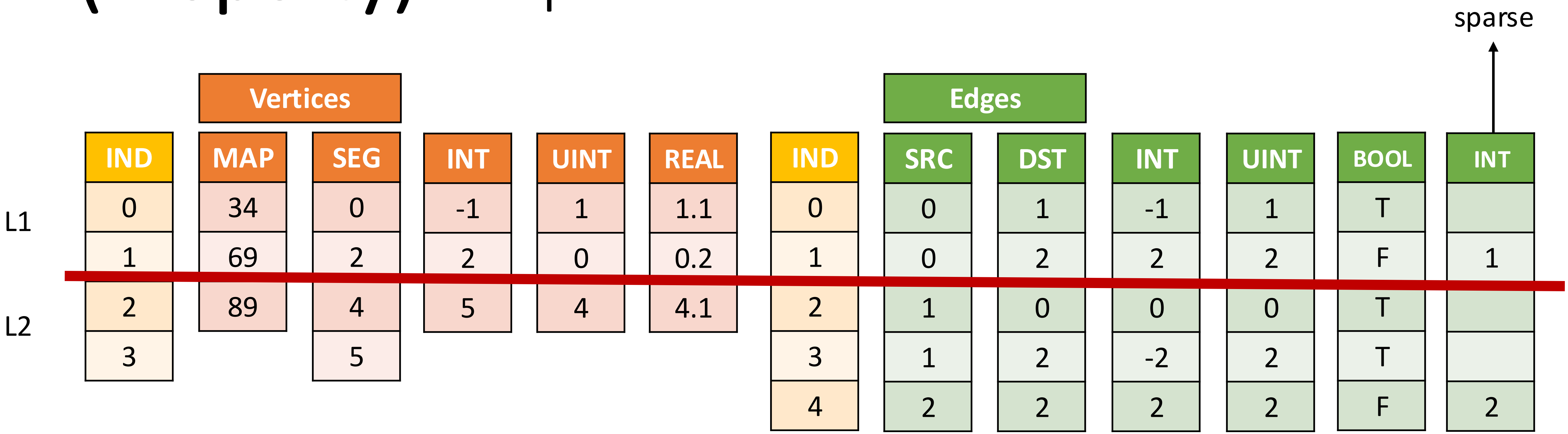
$[SEG[u]..< SEG[u+1]]$

$\leftarrow e$



- 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



- 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.

NJIT New Jersey Institute of Technology

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

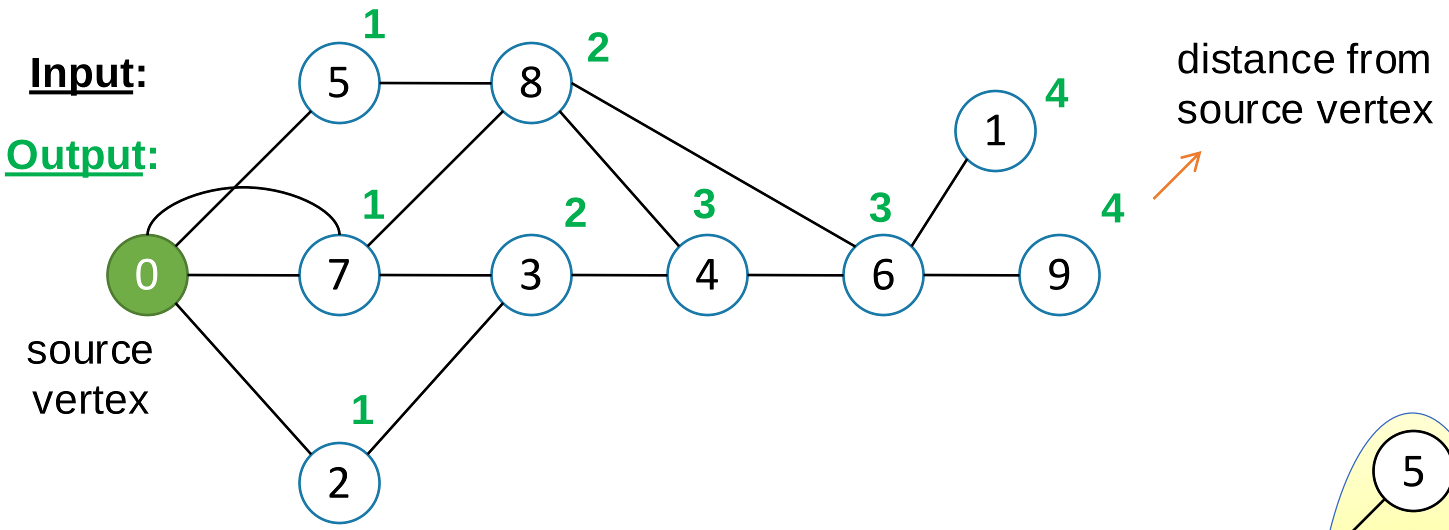
General Information

- 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.

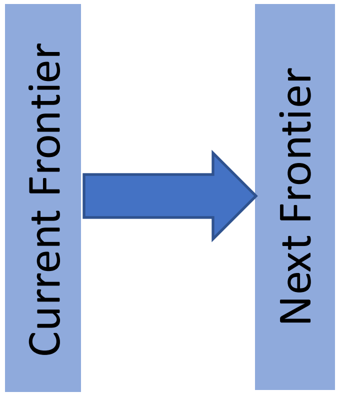
Single Locale Parallel BFS (version 1.0)

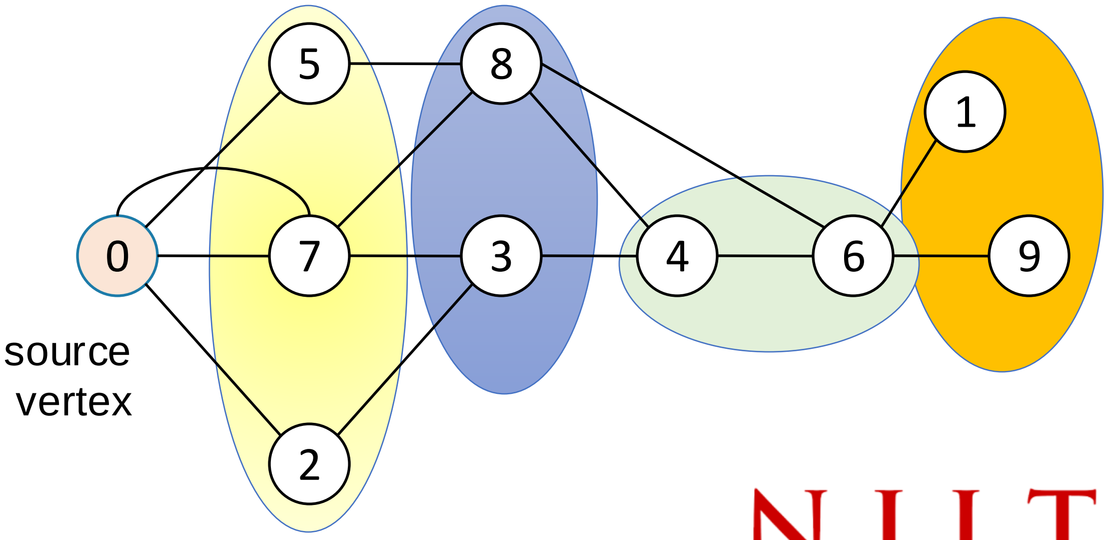
Input:

Output:



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



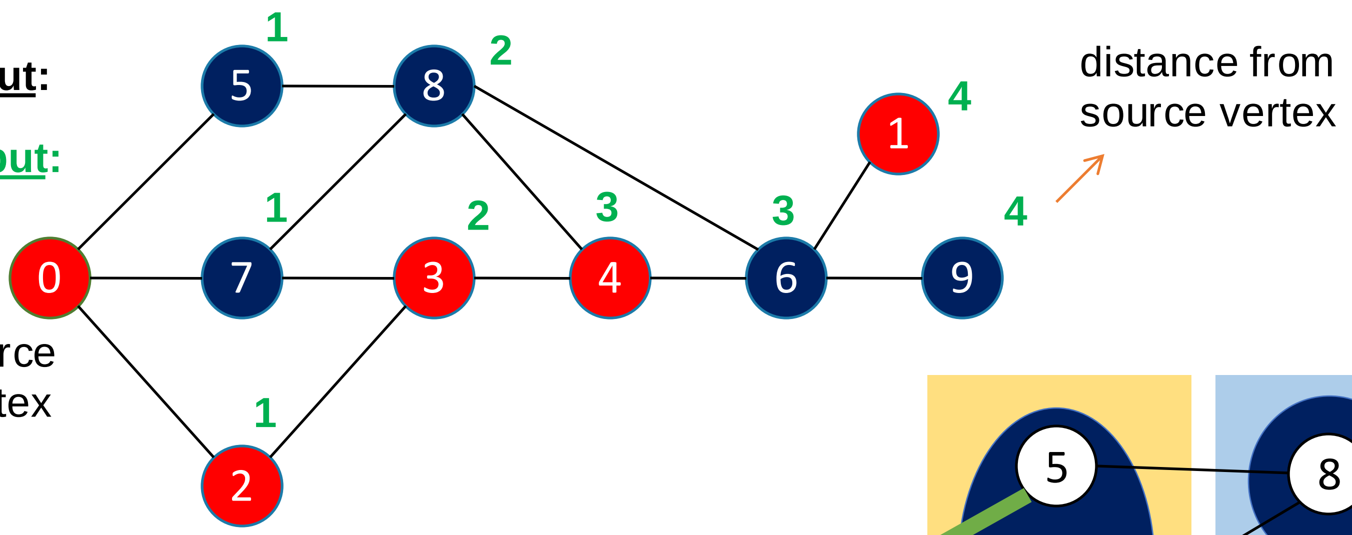


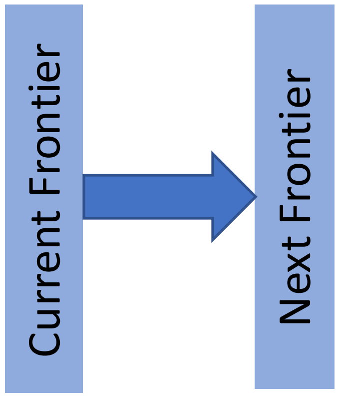
Multilocale Parallel BFS (version 1.5)

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.

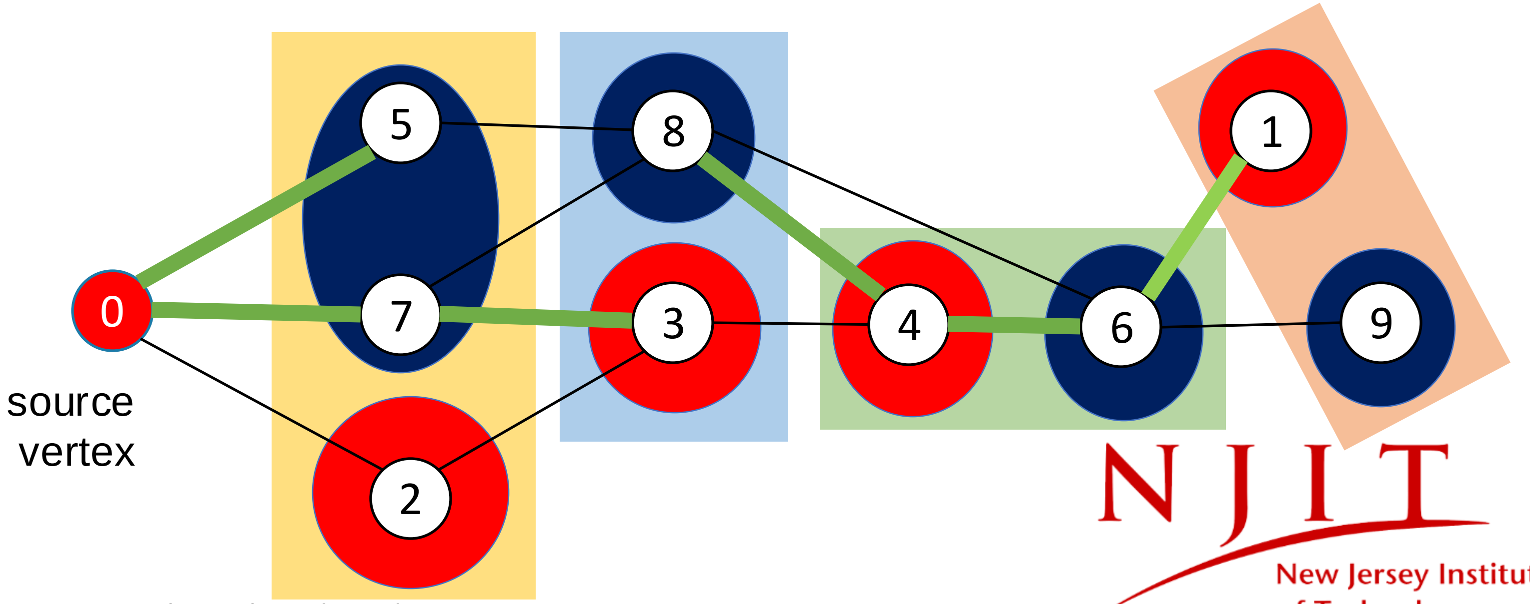
Input:

Output:



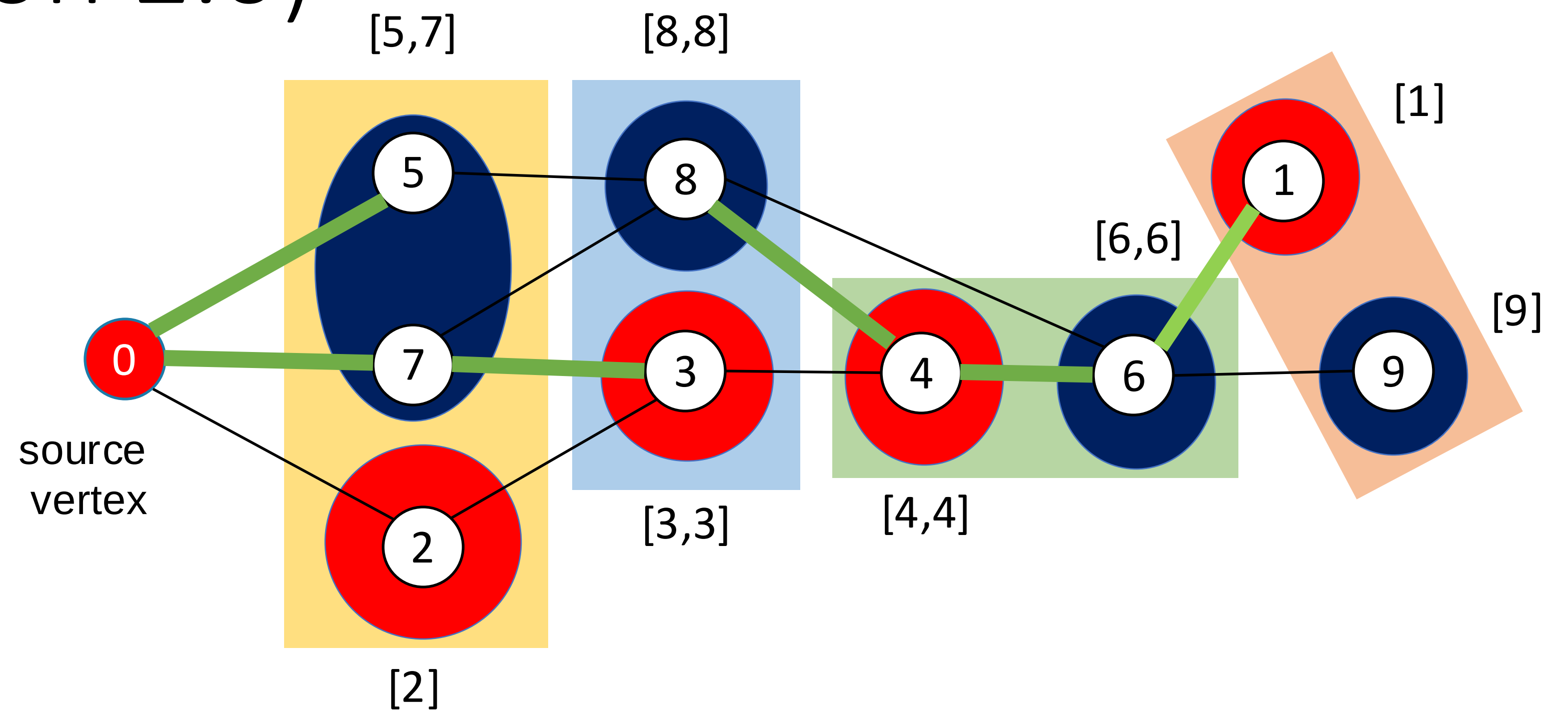


Output: D = [0, 4, 1, 2, 3, 1, 3, 1, 2, 4]



Any cross-color expansions are writes across the network; fine-grained writes hold up execution, large coarse-grained writes are better.

Multilocale Parallel BFS with Aggregators (version 2.0)



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.

New Jersey Institute of Technology

Multilocale Parallel BFS Version 1.0

- Uses ideas of forward and reversed edges for undirected graphs. For example, u—v is stored in SRC and DST and v—u is stored in SRCr and DSTr.
- Use the “old” distributed bag to expand frontiers.

Expand frontier based of forward-edges

Expand frontier based of reversed-edges

```
while (numCurF>0) {
    coforall loc in Locales  with (ref SetNextF,+ reduce topdown, + reduce bottomup, ref root, ref src, ref depth) {
       on loc {
           ref srcf=src;
           ref df=dst;
           ref nf=nei;
           ref sf=start_i;

           ref srcfR=srcR;
           ref dfR=dstR;
           ref nfR=neiR;
           ref sfR=start_iR;

           var edgeBegin=src.localSubdomain().lowBound;
           var edgeEnd=src.localSubdomain().highBound;
           var vertexBegin=src[edgeBegin];
           var vertexEnd=src[edgeEnd];
           var vertexBeginR=srcR[edgeBegin];
           var vertexEndR=srcR[edgeEnd];

           var switchratio=(numCurF:real)/nf.size:real;
           if (switchratio<GivenRatio) {//top down
               topdown+=1;
               forall i in SetCurF with (ref SetNextF) {
                  if ((xlocal(i,vertexBegin,vertexEnd)) || (LF==0)) {// current edge has the vertex
                      var    numNF=nf[i];
                      var    edgeId=sf[i];
                      var nextStart=max(edgeId,edgeBegin);
                      var nextEnd=min(edgeEnd,edgeId+numNF-1);
                      ref NF=df[nextStart..nextEnd];
                      forall j in NF with (ref SetNextF){
                             if (depth[j]==-1) {
                                   depth[j]=cur_level+1;
                                   SetNextF.add(j);
                             }
                      }
                  }

                  if ((xlocal(i,vertexBeginR,vertexEndR)) || (LF==0))  {
                      var    numNF=nfR[i];
                      var    edgeId=sfR[i];
                      var nextStart=max(edgeId,edgeBegin);
                      var nextEnd=min(edgeEnd,edgeId+numNF-1);
                      ref NF=dfR[nextStart..nextEnd];
                      forall j in NF with (ref SetNextF)  {
                             if (depth[j]==-1) {
                                   depth[j]=cur_level+1;
                                   SetNextF.add(j);
                             }
                      }
                  }
```

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.

```
while (numCurF>0) {
    coforall loc in Locales  with (ref SetNextF,+ reduce topdown, + reduce bottomup, ref root, ref src, ref depth) {
        on loc {
            ref srcf=src;
            ref df=dst;
            ref nf=nei;
            ref sf=start_i;

            ref srcfR=srcR;
            ref dfR=dstR;
            ref nfR=neiR;
            ref sfR=start_iR;

            var edgeBegin=src.localSubdomain().lowBound;
            var edgeEnd=src.localSubdomain().highBound;
            var vertexBegin=src[edgeBegin];
            var vertexEnd=src[edgeEnd];
            var vertexBeginR=srcR[edgeBegin];
            var vertexEndR=srcR[edgeEnd];

            var switchratio=(numCurF:real)/nf.size:real;
            if (switchratio<GivenRatio) {//top down
                topdown+=1;
                forall i in SetCurF with (ref SetNextF) {
                    if ((xlocal(i,vertexBegin,vertexEnd)) || (LF==0)) {// current edge has the vertex
                        var     numNF=nf[i];
                        var     edgeId=sf[i];
                        var nextStart=max(edgeId,edgeBegin);
                        var nextEnd=min(edgeEnd,edgeId+numNF-1);
                        ref NF=df[nextStart..nextEnd];
                        forall j in NF with (ref SetNextF){
                                if (depth[j]==-1) {
                                    depth[j]=cur_level+1;
                                    SetNextF.add(j);
                                }
                        }
                    }
```

Expand frontier based of symmetrized edges

New Jersey Institute of Technology

Multilocale Parallel BFS Version 2.0

Locale parallelism

Parallelism per-locale

Maintaining locality

Neighborhood expansion with aggregation

Aggregator per-task

```
while true {
    var pending_work:bool;
    coforall loc in Locales with(|| reduce pending_work, ref depth, ref frontier_sets) {
        on loc {
            var src_low = src.localSubdomain().low;
            var src_high = src.localSubdomain().high;
            forall u in frontier_sets[frontier_sets_idx] with (|| reduce pending_work, var frontier_agg = new ListDstAggregator(int)) {
                var adj_list_start = seg[u];
                var num_neighbors = seg[u+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 = dst.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 lc in locs do frontier_agg.copy(lc.id, v);
                        }
                    }
                }
            } //end forall
            frontier_sets[frontier_sets_idx].clear();
        } // end on loc
    }// end coforall loc
    if !pending_work {
        break;
    }
    cur_level += 1;
    frontier_sets_idx = (frontier_sets_idx + 1) % 2;
}// end while
return "success";
```

Multilocale BFS Communication Volume Heatmap

delaunayn20	get			put		
locale	1.0	1.5	2.0	1.0	1.5	2.0
0	15672640	7873842	639827	5629422	2749193	138070
1	15834332	7939017	687156	1952226	1016946	127936
2	15715554	7722659	226754	1942839	962031	45217
3	15817879	7723971	226880	1951313	962201	45060
4	15964559	7724880	226691	1961552	962199	51217
5	15739226	7726504	230024	1940688	962439	52714
6	15569450	7727678	229096	1925536	962680	51977
7	15341933	7736094	225083	1904757	963418	48413

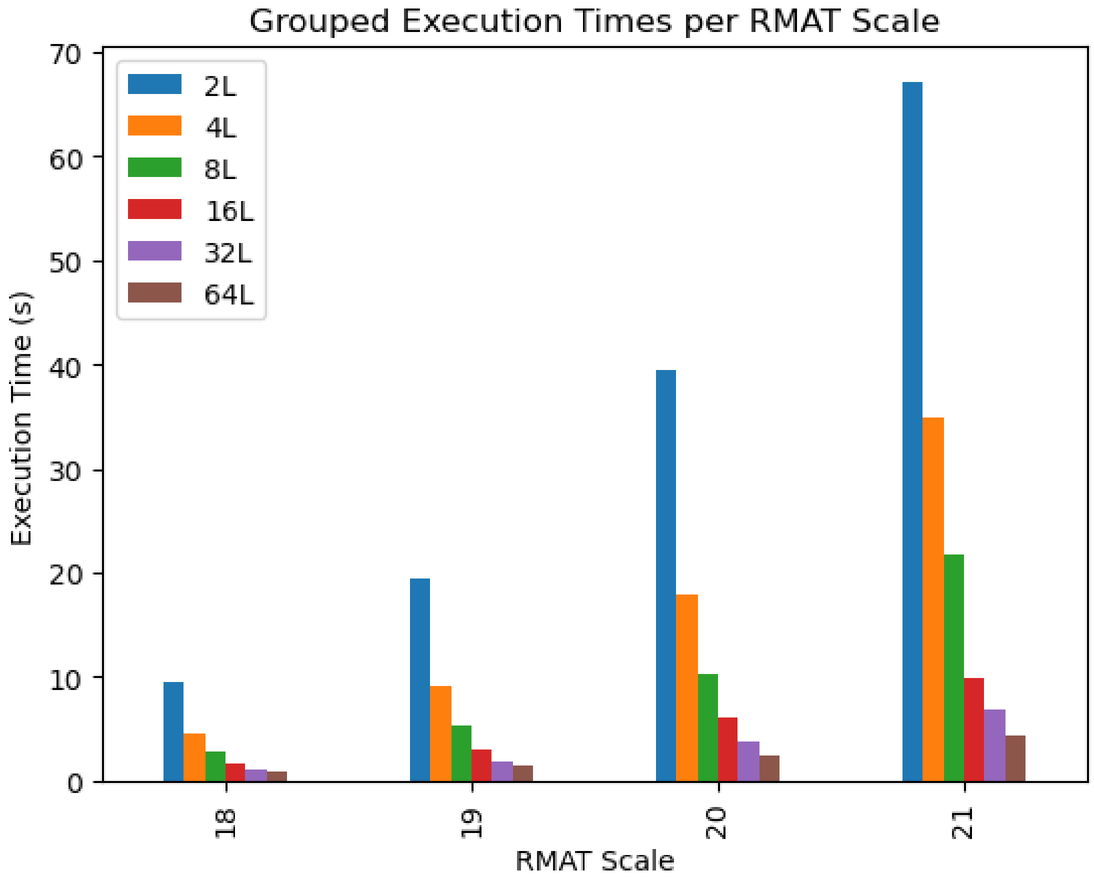
1.0: 84 seconds (HPEC 21')
2.0: 3.36 seconds

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.

New Jersey Institute of Technology

2.0 BFS Scalability



Speed-Up Over 2 Locales

	4L	8L	16L	32L	64L
18	2.11	3.43	5.87	8.10	9.66
19	2.14	3.69	6.35	10.28	13.04
20	2.20	3.84	6.41	10.60	15.90
21	1.93	3.09	6.84	9.86	15.56

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

NJIT New Jersey Institute of Technology

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.

NJIT
New Jersey Institute of Technology

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.

NJIT
New Jersey Institute of Technology

Thank You ☺

Questions?