Exploratory Large Scale Graph Analytics in Arkouda

Zhihui Du (Presenter)
Oliver Alvarado Rodriguez, David A. Bader
Michael Merrill and William Reus

αρκούδα
massive scale
data science

CHAPEL

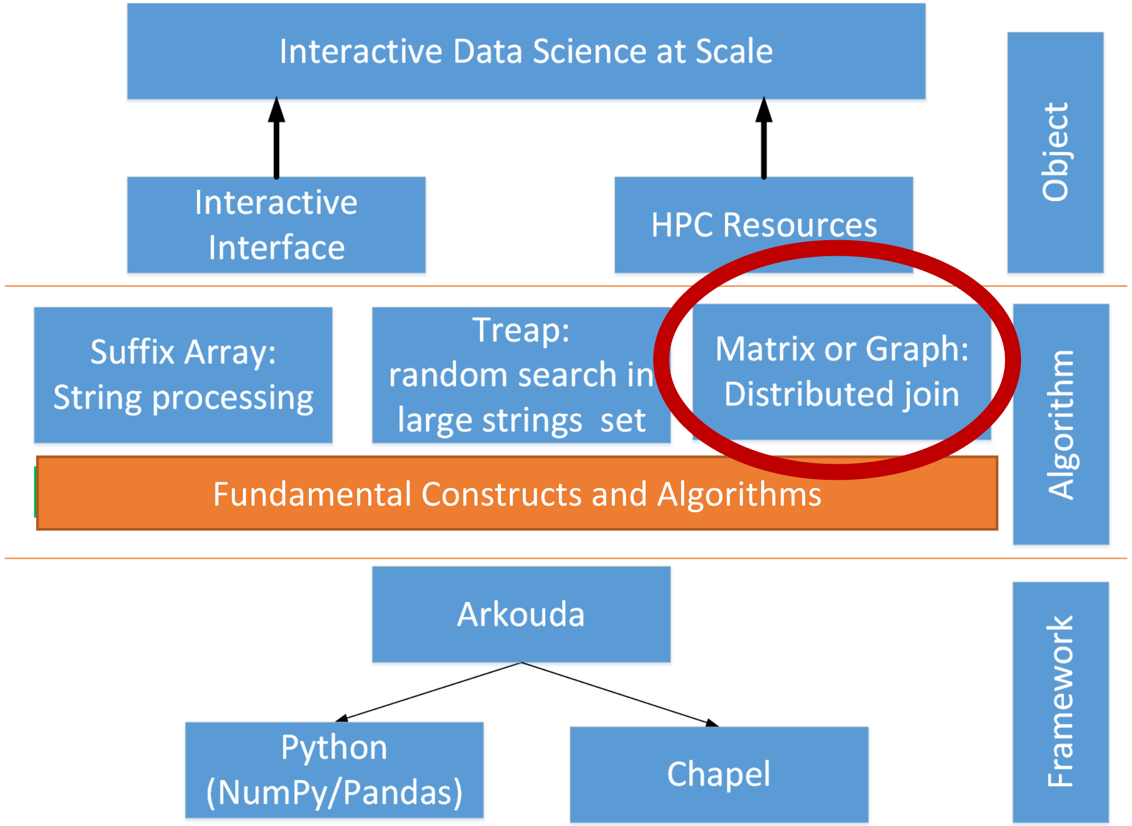
CHIUW 2021

NJIT
New Jersey Institute of Technology

This research is supported by NSF grant CCF-2109988

Overview of the Complete Research

- Objective:
  - One-stop solution for non-HPC users to exploit massive data sets.
- Research focus:
  - Data structures and algorithms
- Framework:
  - Arkouda



NJIT New Jersey Institute of Technology

Why Arkouda?

Bill Reus' CHIUW 2020 Keynote
https://chapel-lang.org/CHIUW/2020/Reus.pdf

We want some of our Data Scientists to drive an F22!

Flexibility+Capability

Jupyter allows Data Scientists to drive a cool plane!

A vital system growing with your need

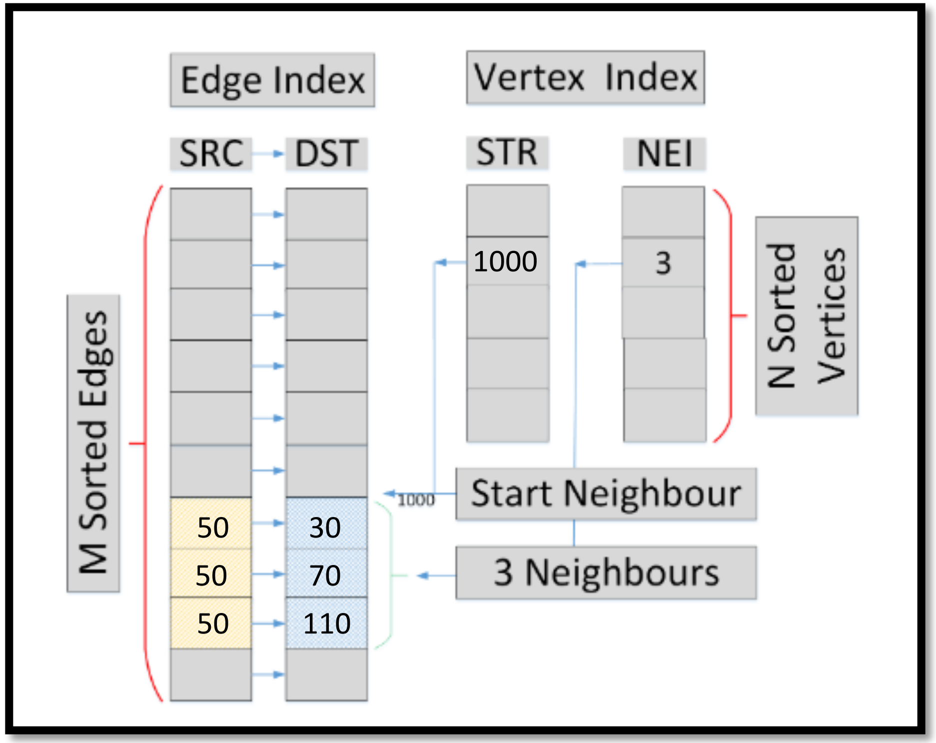
Image From Mike Merrill's CHIUW 2019 Talk
https://chapel-lang.org/CHIUW/2019/Merrill.pdf

Large-Scale Graph Analytics in Arkouda

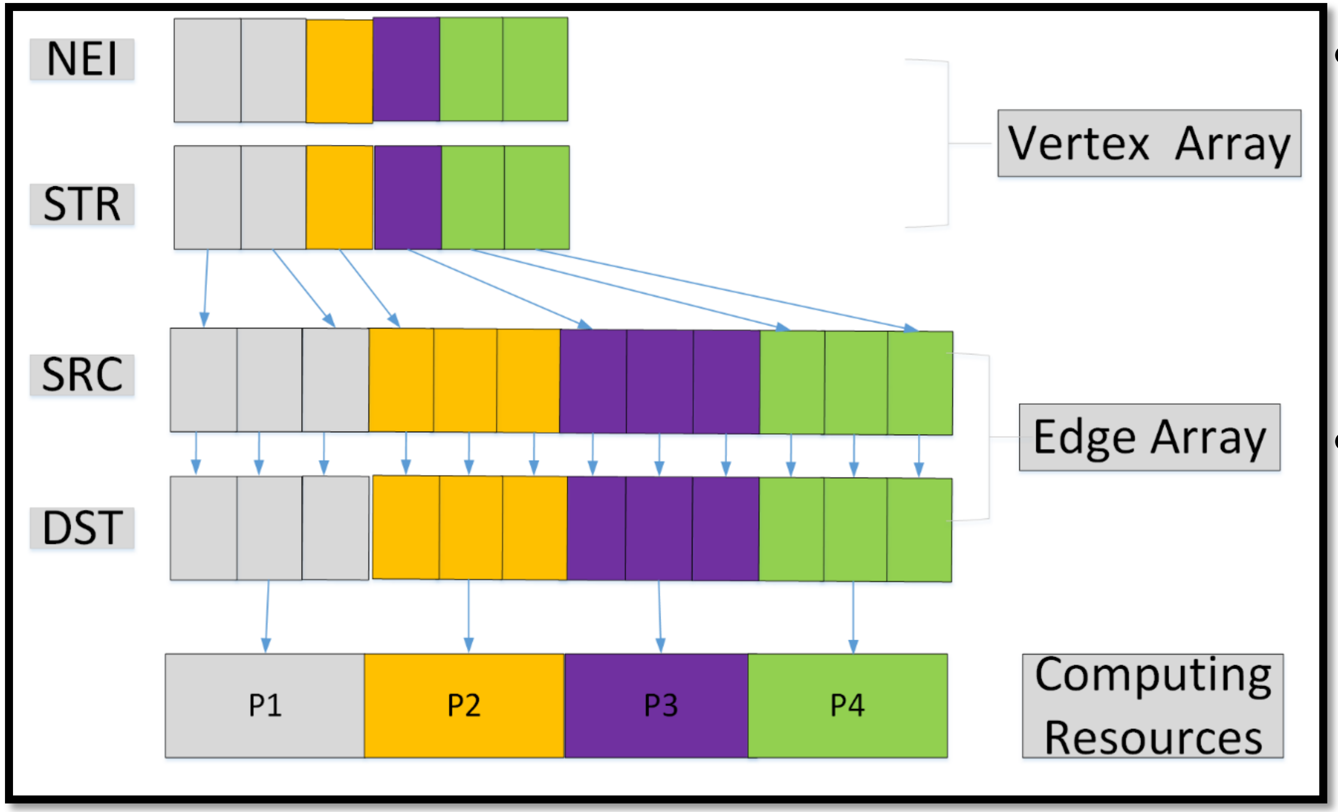
- Why graph and what is the challenge
  - Graph is a powerful tool to represent and solve widely available problems
  - Real-world graphs have become very large
    - billion or trillions of edges, regular computing at a local laptop level becomes more difficult, time consuming, and possibly even impossible!
- What we have done
  - **A double-index based data structure**
  - **Parallel and distributed (multi-locale) breadth-first search algorithm**

Double-Index Graph Data Structure

- Advantage
  - O(1) time complexity
    - Locate specific vertex from given edge ID
    - Locate adjacency list from given vertex ID
- Compared with CSR (compressed sparse row)
  - Similarity
    - Value array <-> SRC/DST, array size is NNZ
    - column index <-> STR , array size is |V|
    - row index <-> NEI, array size is |V|+1 and |V|
  - Difference
    - We can search from edge ID to vertex ID, CSR cannot
    - We need a bit more memory (another NNZ array) than CSR



Support Edge-Oriented Graph Partition



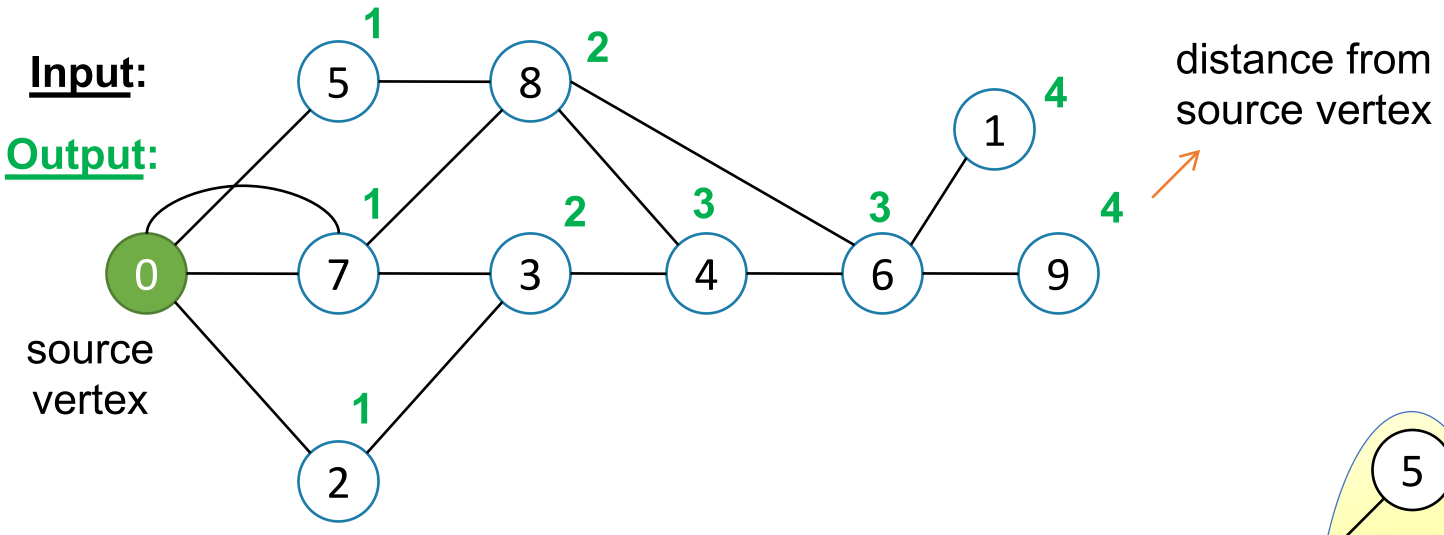
- Imbalance in vertex's degree
  - Power law degree distribution of real-world graphs
    - One vertex can have very different number edges
- Edge oriented partition
  - Edge array is much larger than vertex array
    - Align vertices to corresponding edge
    - load balancing

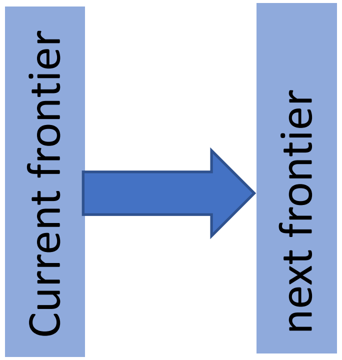
New Jersey Institute of Technology

Breadth-First Search (BFS) Problem

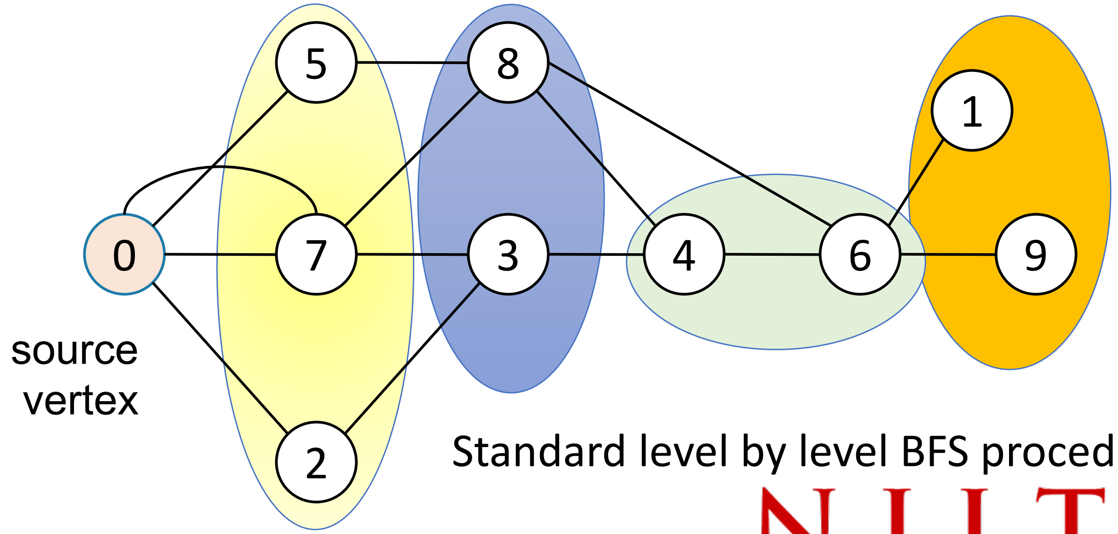
Input:

Output:





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



Standard level by level BFS procedure

High Level Multi-Locale BFS Algorithm

```
Input: A graph G and the starting vertex root
Output: An array depth to show the different visiting level for each vertex
depth = −1 // initialize the visiting level of all the vertices
depth[root] = 0 // set starting vertex's level is 0
cur_level = 0 //set current level
SetCurF = new DistBag(int, Locales) // allocate a distributed bag to hold vertices in the
    current frontier
SetNextF = new DistBag(int, Locales) // allocate another bag to hold vertices in the next
    frontier
SetCurF.add(root) //insert the starting vertex into the current vertices bag
while (!SetCurF.isEmpty()) do
    coforall (loc in Locales ) do
        // parallel search on each locale
        forall (i in SetCurF ) do
            if (i is on current locale) then
                SetNeighbour = {k|k is the neighbour of i}
                forall (j in SetNeighbour) do
                    if (depth[j] == −1) then
                        SetNextF.add(j)
                        depth[j] = current_level + 1
                    end
                end
            end
        end
    end
    SetCurF <=> SetNextF // exchange values
    SetNextF.clear()
    current_level+ = 1;
end
return depth
```

High level data structure

Distributed parallel

Parallel next frontier search

Parallel new vertices insert

NJIT New Jersey Institute of Technology

Low Level Multi-Locale BFS Algorithm

```
Input: A graph G and the starting vertex root
Output: An array depth to show the different visiting level for each vertex
depth = −1 // initialize the visiting level of all the vertices
depth[root] = 0 // set starting vertex's level is 0
cur_level = 0 //set current level
Create distributed array curFAry to hold current frontier of each locale
Create distributed array recvAry to receive expanded vertices from other locales
put root into curFAry
while (!curFAry.isEmpty()) do
    coforall (loc in Locales ) do
        create SetNextFLocal to hold expanded vertices owned by current locale
        create SetNextFRemote to hold expanded vertices owned by other locales
        myCurF ←current locale's frontier in curFAry and then clear curFAry
        coforall (i in myCurF ) do
            SetNeighbour = {k|k is the neighbour of i}
            forall (j in SetNeighbour) do
                if (depth[j] == −1) then
                    if (j is local) then
                        SetNextFLocal.add(j)
                    end
                    else
                        SetNextFRemote.add(j)
                    end
                    depth[j] = current_level + 1
                end
            end
        end
        if (!SetNextFRemote.isEmpty()) then
            scatter elements in SetNextRemote to recvAry
        end
        if (!SetNextFLocal.isEmpty()) then
            move elements in SetNextLocal to curFAry
        end
    end
    coforall (loc in Locales ) do
        curFAry ← collect elements from recvAry
    end
    current_level+ = 1
end
return depth
```

Low level data structure

Distributed parallel execution

Parallel next frontier search

Parallel new vertices insert

Explicit global communication

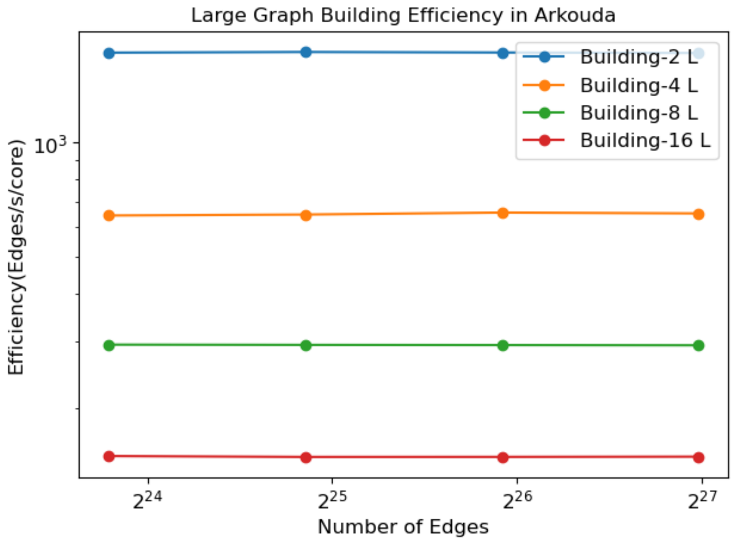
NJIT New Jersey Institute of Technology

Datasets for Experiment (Sparse graphs)

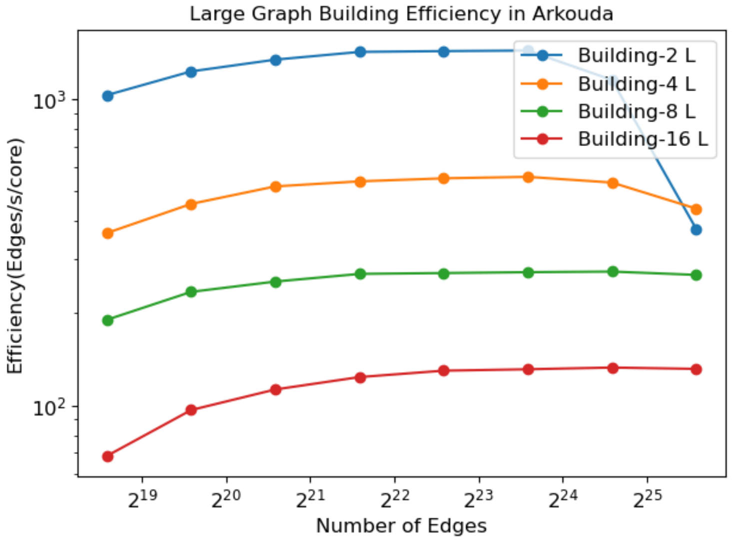
Name	Vertices	Edges	Weighted	CCs	Biggest CC Size	Diameter(≥)
delaunay_n17	131072	393176	0	1	131072	163
delaunay_n18	262144	786396	0	1	262144	226
delaunay_n19	524288	1572823	0	1	524288	309
delaunay_n20	1048576	3145686	0	1	1048576	442
delaunay_n21	2097152	6291408	0	1	2097152	618
delaunay_n22	4194304	12582869	0	1	4194304	861
delaunay_n23	8388608	25165784	0	1	8388608	1206
delaunay_n24	16777216	50331601	0	1	16777216	1668
rgg_n_2_21_s0	2097148	14487995	0	4	2097142	1151
rgg_n_2_22_s0	4194301	30359198	0	2	4194299	1578
rgg_n_2_23_s0	8388607	63501393	0	4	8388601	2129
rgg_n_2_24_s0	16777215	132557200	0	1	16777215	3009
kron_g500-logn18	210155	10583222	1	8	210141	4
kron_g500-logn19	409175	21781478	1	27	409123	4
kron_g500-logn20	795241	44620272	1	45	795153	4
kron_g500-logn21	1544087	91042010	1	94	1543901	4

New Jersey Institute of Technology

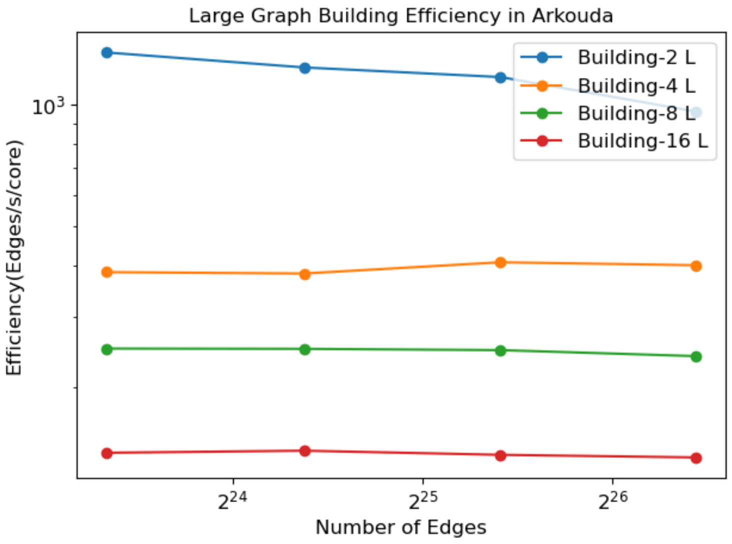
Graph Building in Arkouda



RMAT



Delaunay



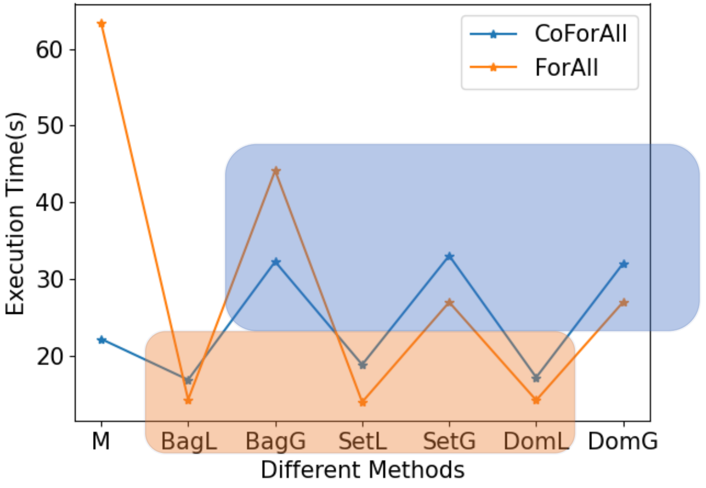
Kron

Parallel data reading/generating+graph sorting

Almost the same building efficiency/resource efficiency for different number of edges on the same resource

New Jersey Institute of Technology

Results of different BFS Variants

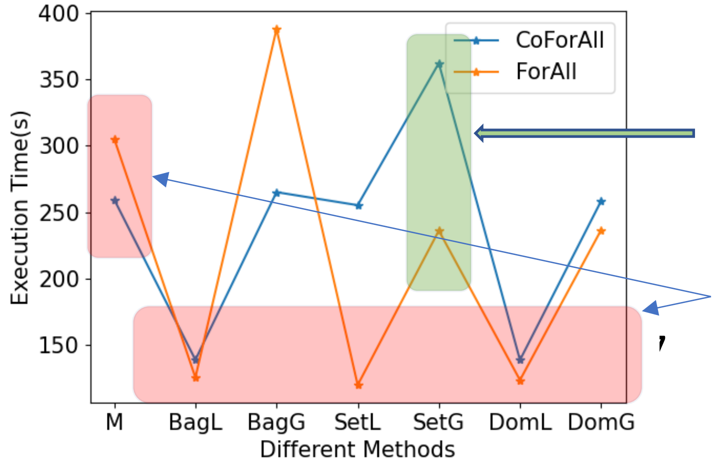


Delaunay 17

- High level data structure
  - Distbag, set, and domain
- Parallel construct
  - forall/coforall

redundant calculation without idle threads

no redundant calculation with idle threads



Delaunay 20

Different parallel constructures can affect performance

High level algorithm can compete with low level algorithm under the same algorithm framework

NJIT New Jersey Institute of Technology

Optimization results

Employ reverse Cuthill-Mckee (RCM) algorithm as a preprocessing step

Graph	Parallel Construct	RCM	M	BagL	BagG	SetL	SetG	DomL	DomG
delaunay_n17	CoForall	N	22.20	16.87	32.28	18.84	33.05	17.18	32.06
		Y	14.90	14.77	26.68	16.94	29.11	14.42	26.65
	Forall	N	63.42	14.28	44.14	13.97	26.99	14.20	27.02
		Y	24.28	10.85	33.75	12.02	21.85	12.16	21.85
delaunay_n18	CoForAll	N	48.57	33.76	64.58	43.08	70.55	34.25	63.84
		Y	31.08	30.91	55.62	47.10	70.52	32.51	55.58
	ForAll	N	155.39	28.37	87.79	27.59	53.58	28.26	54.07
		Y	37.56	23.37	73.45	25.28	43.52	25.58	44.05
delaunay_n19	CoForAll	N	110.93	68.72	131.04	102.32	156.55	69.08	128.39
		Y	63.77	63.83	114.82	114.05	159.83	62.55	109.56
	ForAll	N	453.23	56.54	175.88	55.62	107.17	56.49	107.56
		Y	69.90	46.23	141.92	49.65	86.68	50.27	86.50
delaunay_n20	CoForAll	N	259.44	139.16	265.08	255.28	361.99	138.98	258.44
		Y	126.62	127.22	231.47	286.72	386.11	133.12	229.45
	ForAll	N	305.01	125.89	387.61	120.19	236.20	123.91	236.66
		Y	172.16	92.87	293.59	99.46	176.49	101.05	176.03

Low level algorithm results

High level algorithm results

Conclusion

- Arkouda (Python+ Chapel) can be used to handle large graph analytics with two advantages:
  - High productivity and high performance
- Chapel based high level parallel graph kernel algorithm development can achieve high performance
  - even better than low level message passing method for our case
- First step to evaluate the feasibility and performance of Arkouda-based large graph analytics
  - more graph algorithms and more optimizations in future work

New Jersey Institute of Technology

Acknowledgement

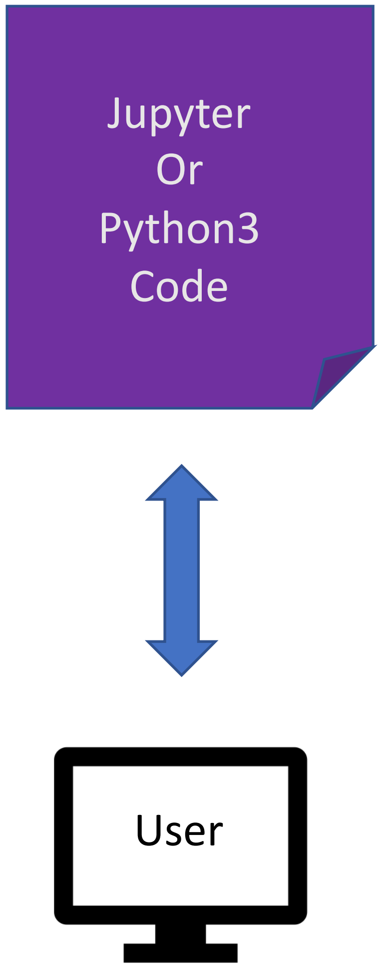
We appreciate the help from Brad Chamberlain, Elliot Joseph Ronaghan, Engin Kayraklioglu, David Longnecker and the Chapel community when we integrated the algorithms into Arkouda. This research was funded in part by NSF grant number CCF-2109988.

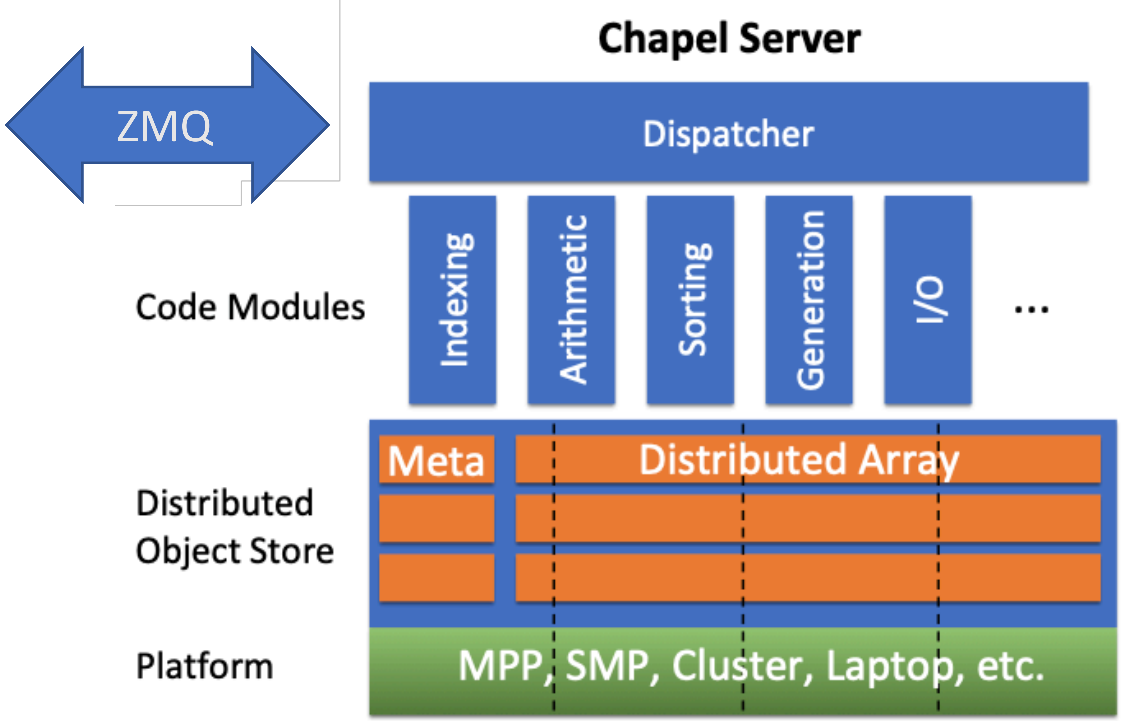
NJIT New Jersey Institute of Technology

Thank You!

Q&A

Typical Environment Set-Up





Python3 Implementation:

- Pdarray class
- Rely on Python to reduce complexity
- Integrate with and use NumPy

Server Implementation:

- High-level language with C-comparable performance
- Great parallelism handling
- Great distributed array support
- Portable code: laptop --> HPC

Where can I get it?:
Image: https://chapel-lang.org/CHIUW/2020/Reus.pdf
Software: https://github.com/mhmerrill/arkouda
Our Contribution: https://github.com/Bader-Research/arkouda/tree/streaming

Arkouda: Maximize the benefit of Data Science

- Barriers to exploit data science
  - Interface barrier: low level programming->high level programming
  - Resource barrier: PC resources->cloud/supercomputing resources
- What is the challenging problem?
  - Interactive (enough flexibility) + Large-scale analytics (enough capability)