Arkouda (αρκούδα):
Interactive Supercomputing for Data Analytics Made Possible by Chapel

Michael Merrill

SIAM PP-22
MS34: Achieving Productivity at Scale with Chapel in User Applications
February 24, 2022

Outline

- A quick teaser example of Arkouda to engage people.
- Motivation for something like Arkouda
- Why Chapel?
- Hero result "big sort" graph and why it is important to us.

Arkouda Startup

1) In terminal:

```
> arkouda_server –nl 96

server listening on hostname:port
```

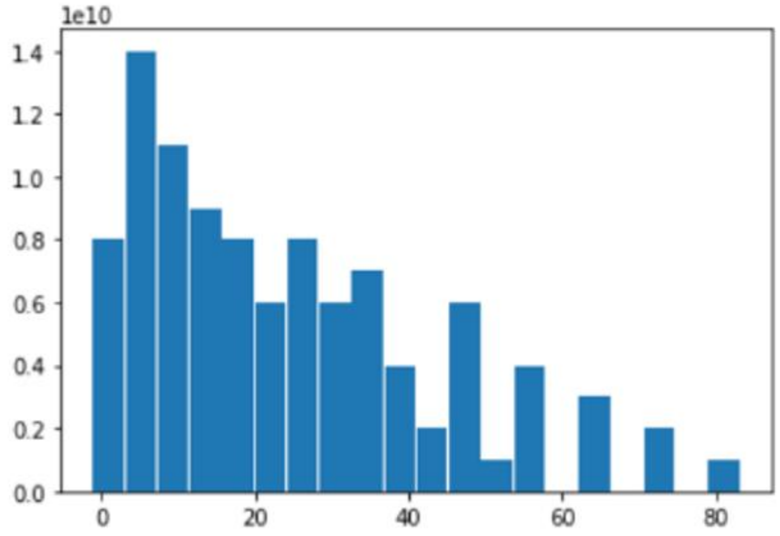
2) In Jupyter:

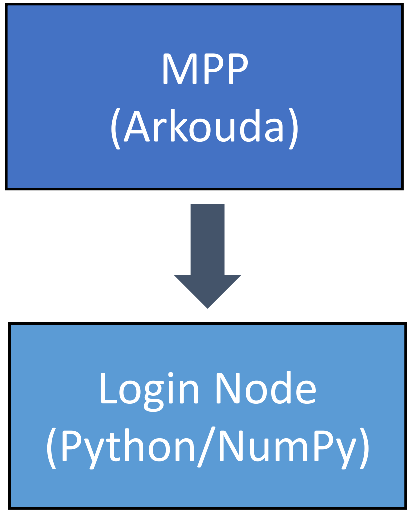
```
In [2]: import arkouda as ak
        ak.connect(hostname, port)

4.2.5
psp =  tcp://nid00104:5555
connected to tcp://nid00104:5555
```

Toy Workflow

In [9]:
```
A = ak.randint(0, 10, 10**11)
B = ak.randint(0, 10, 10**11)
C = A * B
hist = ak.histogram(C, 20)
Cmax = C.max()
Cmin = C.min()
```
executed in 3.96s, finished 13:45:28 2019-09-12

In [10]:
```
bins = np.linspace(Cmin, Cmax, 20)
_ = plt.bar(bins, hist.to_ndarray(), width=(Cmax-Cmin)/20)
```
executed in 193ms, finished 13:45:28 2019-09-12





More complex Arkouda example

R-MAT Graph Generator

```python
#!/usr/bin/env python3
import arkouda as ak

# generate rmat graph edge-list as two pdarrays
def gen_rmat_edges(lgNv, Ne_per_v, p, perm=False):
    # number of vertices
    Nv = 2**lgNv
    # number of edges
    Ne = Ne_per_v * Nv
    # probabilities
    a = p
    b = (1.0 - a)/ 3.0
    c = b
    d = b
    # init edge arrays
    ii = ak.ones(Ne,dtype=ak.int64)
    jj = ak.ones(Ne,dtype=ak.int64)
    # quantites to use in edge generation loop
    ab = a+b
    c_norm = c / (c + d)
    a_norm = a / (a + b)
    # generate edges
    for ib in range(1,lgNv):
        ii_bit = (ak.randint(0,1,Ne,dtype=ak.float64) > ab)
        jj_bit = (ak.randint(0,1,Ne,dtype=ak.float64) > (c_norm * ii_bit + a_norm * (~ ii_bit)))
        ii = ii + ((2**(ib-1)) * ii_bit)
        jj = jj + ((2**(ib-1)) * jj_bit)
    # sort all based on ii and jj using coargsort
    # all edges should be sorted based on both vertices of the edge
    iv = ak.coargsort((ii,jj))
    # permute into sorted order
    ii = ii[iv] # permute first vertex into sorted order
    jj = jj[iv] # permute second vertex into sorted order
    # to premute/rename vertices
    if perm:
        # generate permutation for new vertex numbers(names)
        ir = ak.argsort(ak.randint(0,1,Nv,dtype=ak.float64))
        # renumber(rename) vertices
        ii = ir[ii] # rename first vertex
        jj = ir[jj] # rename second vertex
    #
    # maybe: remove edges which are self-loops???
    #
    # return pair of pdarrays
    return (ii,jj)
```

-:--- connected_components.py Top (20,24) (Python)

BFS

Connected Components

```python
# src and dst pdarrays hold the edge list
# seeds pdarray with starting vertices/seeds
def bfs(src,dst,seeds,printLayers=False):
    # holds vertices in the current layer of the bfs
    Z = ak.unique(seeds)
    # holds the visited vertices
    V = ak.unique(Z) # holds vertices in Z to start with
    # frontiers
    F = [Z]
    while Z.size != 0:
        if printLayers:
            print("Z.size = ",Z.size," Z = ",Z)
        fZv = ak.in1d(src,Z) # find src vertex edges
        W = ak.unique(dst[fZv]) # compress out dst vertices to match and make them unique
        Z = ak.setdiff1d(W,V) # subtract out vertices already visited
        V = ak.union1d(V,Z) # union current frontier into vertices already visited
        F.append(Z)
    return (F,V)

# src pdarray holding source vertices
# dst pdarray holding destination vertices
# printCComp flag to print the connected components as they are found
# edges needs to be symmetric/undirected
def conn_comp(src, dst, printCComp=False, printLayers=False):
    unvisited = ak.unique(src)
    if printCComp: print("unvisited size = ", unvisited.size, unvisited)
    components = []
    while unvisited.size > 0:
        # use lowest numbered vertex as representative vertex
        rep_vertex = unvisited[0]
        # bfs from rep_vertex
        layers,visited = bfs(src,dst,ak.array([rep_vertex]),printLayers)
        # add verticies in component to list of components
        components.append(visited)
        # subtract out visited from unvisited vertices
        unvisited = ak.setdiff1d(unvisited,visited)
        if printCComp: print("  visited size = ", visited.size, visited)
        if printCComp: print("unvisited size = ", unvisited.size, unvisited)
    return components

ak.connect(server="localhost", port=5555)
(ii,jj) = gen_rmat_edges(20, 2, 0.03, perm=True)
src = ak.concatenate((ii,jj))# make graph undirected/symmetric
dst = ak.concatenate((jj,ii))# graph needs to undirected for connected components to work
components = conn_comp(src, dst, printCComp=False, printLayers=False) # find components
print("number of components = ",len(components))
print("representative vertices = ",[c[0] for c in components])
ak.shutdown()
```

-:--- connected_components.py Bot (58,0) (Python)

Why Arkouda?

- Born out of the need to fill some gaps
  - We needed agility at scale.
  - Huge data set exploration and characterization.
- What was needed that didn’t exist?
  - Scalability and performance available from Python because Python is the “new bash” for data science.
  - Speed/Ease of development directed by the needs and implemented by a very small team.

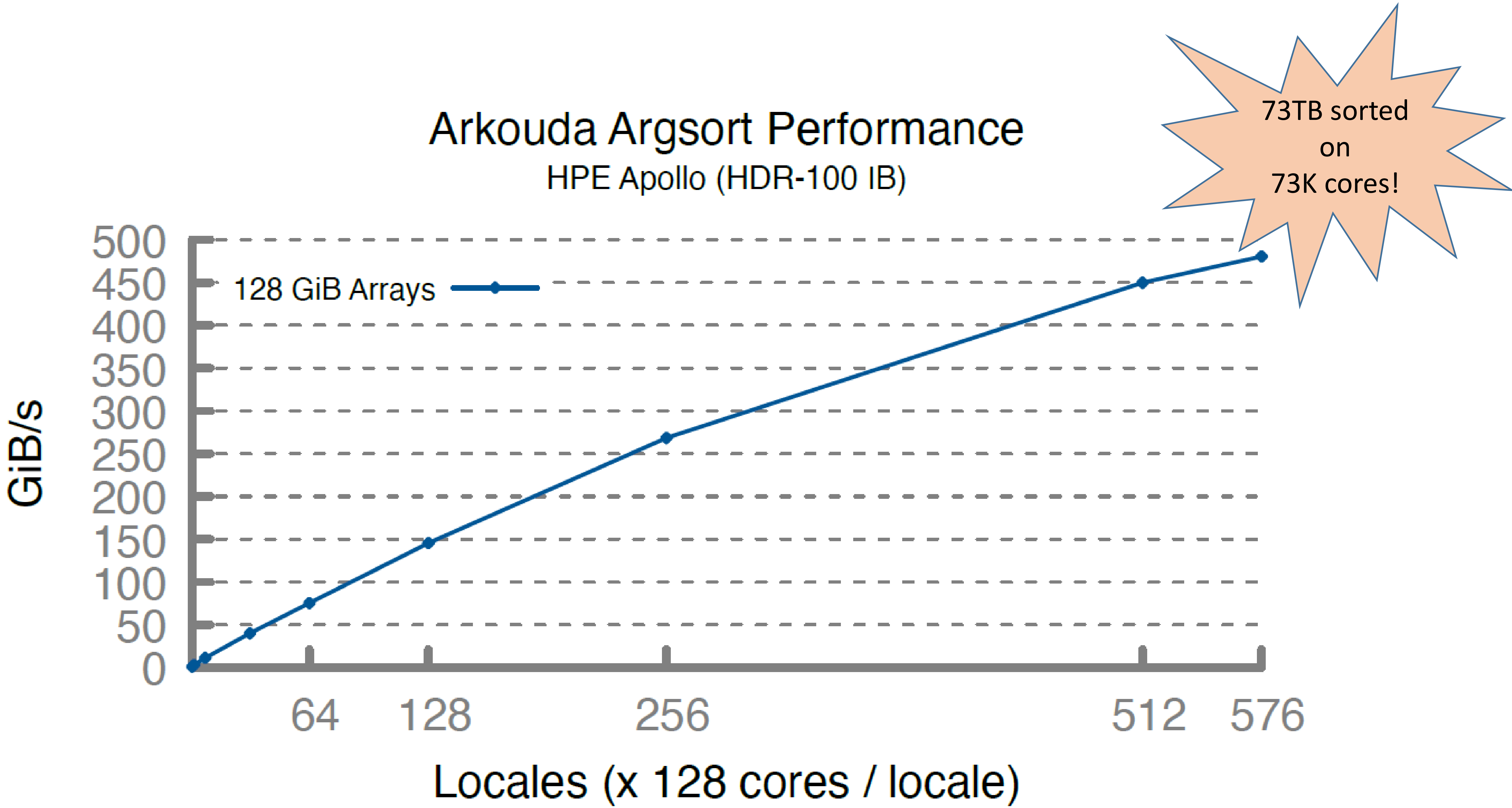
Chapel Is Unique

Why Chapel? -- How did Chapel benefit Arkouda development?

- Productivity
  - Parallelism and locality are first-class citizens
  - Multi-resolution parallelism in code – high level for most of the code and lower level when you need it for performance
  - Small Development team originally two people
  - Arkouda server = ~18k lines of code
- Performance
  - Single-threaded comparable to NumPy (C/Fortran)
  - Parallel, distributed comparable to C/OpenMP/MPI
- Portability
  - Develop on laptop, run on supercomputer

Arkouda Sort/GroupBy

- ak.GroupBy! Underlies almost all analyses we conduct
- A lexicographical sort underpins the GroupBy
- We currently use a Least Significant Digit Radix Sort algorithm which is data distribution agnostic.
- Our Radix Sort is ~100 lines of Chapel
- Uses Chapel's multi-resolution parallel approach
- Incremental optimization by "lowering" loops along with the creation/addition of aggregation capability
- Great scalability!



Arkouda Argsort Performance
HPE Apollo (HDR-100 IB)
73TB sorted on 73K cores!
128 GiB Arrays
GiB/s
500
450
400
350
300
250
200
150
100
50
0
64
128
256
512
576
Locales (x 128 cores / locale)

Arkouda's vision

Load Terabytes of data…

… into a familiar, interactive UI …

… where standard data science operations …

… execute within the human thought loop …

… and interoperate with optimized libraries.

Arkouda: an HPC shell for data science

- Chapel backend (server)
- Jupyter/Python frontend (client)
- NumPy- and Pandas-like API

A New (Old) Perspective on HPC

Not Just This

But Also This

Thanks!

αρκούδα
massive scale
data science

- Dr. William (Bill) Reus (primary collaborator)
- Arkouda Team
- Chapel Team

- Arkouda on GitHub https://github.com/Bears-R-Us/arkouda