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:

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

   4.2.5
   psp = tcp://nid00104:5555
   connected to tcp://nid00104:5555
Toy Workflow

**Login Node (Python/NumPy)**

```python
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 13:45:29 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

```python
#!/usr/bin/env python3

import arkouda as ak

# generate random graph edge-list as two pdarrays
def gen_rmat_edges(Nw, Nv, p, perm=False):
    # number of vertices
    Nv = 2**lgNv
    # number of edges
    Nw = Nv * Nv
    # probabilities
    a = p
    b = (1.0 - a) / 3.0
    c = 1.0 / (1.0 + b)
    # init edge arrays
    ii = ak.ones(Nw, dtype=ak.int64)
    jj = ak.ones(Nw, dtype=ak.int64)
    # quantities to use in edge generation loop
    ab = a*b
    c_nrm = c / (c + d)
    # normalize edges
    for iz in range(1, lgNv):
        ii_bit = ak.randbits(0,1,Nw,dtype=ak.int64) > ab
        jj_bit = ak.randbits(0,1,Nw,dtype=ak.int64) > (c_nrm + ii_bit + c_nrm * (~ii_bit))
        ii = ii + (2**(zi-1)) * ii_bit
        jj = jj + (2**(zi-1)) * jj_bit
    # sort all based on ii and jj using coarsen
    all_edges should be sorted based on both vertices of the edge
    iv = ak.coarsen((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 permute/renumber vertices
    if perm:
        # generate permutation for new vertex numbers(names)
        ir = ak.argsort(ak.randbits(0,1,Nv,dtype=ak.int64))
        # renumber(renames) 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)
```

```python
# src and dst pdarrays hold the edge list
# seeds parray with starting vertices/seed
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
    Ne = ak.shape(Z)

    while Z.size > 0:
        if printLayers:
            print(f"Z.size = "{Z.size}, Z = "{Z})
        f2V = ak.in1d(src, Z) # find src vertex edges
        W = ak.unique(dst[f2V]) # 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
    return (F,V)

# src pdarray holding source vertices
# dst pdarray holding destination vertices
# printComp flag to print the connected components as they are found
# edges needs to be symmetric/undirected
def conn_comp(src, dst, printComp=False, printLayers=False):
    # if printComp: 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 vertices in component to list of components
        components.append(visited)
        # subtract out visited from unvisited vertices
        unvisited = ak.setdiff1d(unvisited,visited)
        if printComp:
            print("visited size = ", visited.size, visited)
        if printComp:
            print("unvisited size = ", unvisited.size, unvisited)
    return components

ak.connect("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(ii,jj) # graph needs to be undirected for connected components to work
components = conn_comp(src, dst, printComp=True, printLayers=False) # find connected components
print("number of components = ",len(components))
print("representative vertices = ",[c[0] for c in components])
```
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!
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!

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

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