ARKOUDA: A HIGH-PERFORMANCE DATA ANALYTICS FRAMEWORK

Michelle Strout with Scott Bachman, Brad Chamberlain, Ben McDonald, and Elliot Ronaghan
Cray User Group (CUG)
May 10, 2023
MOTIVATION FOR ARKOUDA

Motivation: Say you have...
...a bunch of Python programmers
...HPC-scale data science problems to solve
...access to HPC systems

How can you enable your Python programmers to solve large problems?
ARKOUDA’S HIGH-LEVEL APPROACH

Arkouda Client
(written in Python)

Arkouda Server (written in Chapel)

 Writes Python code in Jupyter
Invoking NumPy operations
EXAMPLE ARKOUDA CODE

Summing numbers similar to how one would do with NumPy

```python
>>> N = 10**6
>>> A = ak.arange(1, N+1, 1)  # creating a large array on server
>>> print(A.sum())  # compute sum and returning result to the Python client
```

Keeping arrays and results on the server

```python
# Generate two (server-side) arrays of random integers 0-9
>>> B = ak.randint(0, 10, N)
>>> C = ak.randint(0, 10, N)
>>> D = B * C  # multiply them on the server

# Print a small representation of the array
# This does NOT move the array to the client
>>> print(D)
>>> minVal = D.min()  # compute min and max and bring over to Python
>>> maxVal = D.max()
>>> print(minVal, maxVal)
```
Arkouda Design

Python3 Client

ZMQ Socket

Chapel Server

Dispatcher

Indexing

Arithmetic

Sorting

Generation

I/O

Code Modules

Distributed Object Store

Platform

Meta

Distributed Array

MPP, SMP, Cluster, Laptop, etc.

presented by Bill Reus at CHIUW 2020 on May 22, 2020
ARKOUDA DETAILS

- A Python library supporting data science operations at massive scales and interactive rates
  - massive scales = dozens of terabytes
  - interactive rates = operations that run within the human thought loop (i.e., seconds to small numbers of minutes)
  - implemented using a Client-Server model

- Arkouda client library:
  - a normal Python library, written natively in Python
    - available to Python programmers in standard ways (e.g., Jupyter notebooks, Python interpreter)
  - supports a key subset of operations from the standard NumPy and Pandas libraries
    - e.g., numerical operations, reductions, histograms, sorting, groupby, gather/scatter, ...

- Arkouda server back-end:
  - implemented in Chapel
  - key datatype: 1D distributed arrays
Arkouda is a framework for interactive, high performance data analytics

- Users can and have created more complex computations in Python with Arkouda
- Modular configuration and build
- Server written in Chapel, thus can be extended to any parallel/distributed computations
- Open-source: https://github.com/Bears-R-Us/arkouda

Creators, maintainers, and users

- Mike Merrill, Bill Reus, et al., US DOD, created it within about 9 months of part time work in consultation with Brad Chamberlain at Cray/HPE in 2019
- Elliot Ronaghan and Ben McDonald from the Chapel team help support it
- Scott Bachman is a visiting climate scientist from NCAR who has been experimenting with it

Systems it has and is being run on

- ~360 node Cray XC (11,320 cores)
- 576 nodes of an HPE Apollo with HDR-100 IB (73,728 cores of AMD Rome)
- 896 nodes of an HPE Cray EX with Slingshot 11 (114,688 cores of AMD Milan)
- Other systems: 12TB HPE Superdome X, Cheyenne (SGI ICE XA and IB), Summit (IBM Power 9 and Nvidia Tesla)
Some of the reasons given for picking the Chapel programming language

- High-level language with C-comparable performance
- Parallelism is a first-class citizen
- Great distributed array support
- Portable code: from laptop up to supercomputer
- Integrates with [distributed] numeric libraries
- Close to Pythonic (for a statically typed language)
  - provides a gateway for data scientists ready to go beyond Python

```chapel
var D = {1..1000, 1..1000} dmapped Block(...),
A: [D] real;

forall (i,j) in D do
A[i,j] = i + (j - 0.5)/1000;
```
Understanding Physics of Datasets

Many names: Exploratory Data Analysis, Data Wrangling, Data Modeling, etc.

presented by Bill Reus at CHIUW 2020 on May 22, 2020
## Data Science on 50 Billion Records

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Approx. Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read from disk</td>
<td>A = ak.read_hdf()</td>
<td>30-60</td>
</tr>
<tr>
<td>Scalar Reduction</td>
<td>A.sum()</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Histogram</td>
<td>ak.histogram(A)</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Vector Ops</td>
<td>A + B, A == B, A &amp; B</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Logical Indexing</td>
<td>A[B == val]</td>
<td>1 - 10</td>
</tr>
<tr>
<td>Set Membership</td>
<td>ak.in1d(A, set)</td>
<td>1</td>
</tr>
<tr>
<td>Gather</td>
<td>B = Table[A]</td>
<td>4 - 120</td>
</tr>
<tr>
<td>Get Item</td>
<td>print(A[42])</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Sort Indices by Value</td>
<td>I = ak.argsort(A)</td>
<td>15</td>
</tr>
<tr>
<td>Group by Key</td>
<td>G = ak.GroupBy(A)</td>
<td>30</td>
</tr>
<tr>
<td>Aggregate per Key</td>
<td>G.aggregate(B, 'sum')</td>
<td>10</td>
</tr>
</tbody>
</table>

- **A, B are 50 billion-element arrays of 32-bit values**
- **Timings measured on real data**
- **Hardware:** Cray XC40
  - 96 nodes
  - 3072 cores
  - 24 TB
  - Lustre filesystem

---

Presented by Bill Reus at CHIUW 2020 on May 22, 2020
# Arkouda Performance Compared to NumPy on Cray XC (May 2020)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>NumPy 0.75 GB</th>
<th>Arkouda (serial) 0.75 GB 1 core, 1 node</th>
<th>Arkouda (parallel) 0.75 GB 36 cores x 1 node</th>
<th>Arkouda (distributed) 384 GB 36 cores x 512 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>argsort</td>
<td>0.03 GiB/s</td>
<td>0.05 GiB/s</td>
<td>0.50 GiB/s</td>
<td>55.12 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>1.66x</td>
<td>16.7x</td>
<td>1837.3x</td>
</tr>
<tr>
<td>coargsort</td>
<td>0.03 GiB/s</td>
<td>0.07 GiB/s</td>
<td>0.50 GiB/s</td>
<td>29.54 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>2.3x</td>
<td>16.7x</td>
<td>984.7x</td>
</tr>
<tr>
<td>gather</td>
<td>1.15 GiB/s</td>
<td>0.45 GiB/s</td>
<td>13.45 GiB/s</td>
<td>539.52 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>0.4x</td>
<td>11.7x</td>
<td>469.1x</td>
</tr>
<tr>
<td>reduce</td>
<td>9.90 GiB/s</td>
<td>11.66 GiB/s</td>
<td>118.57 GiB/s</td>
<td>43683.00 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>1.2x</td>
<td>12.0x</td>
<td>4412.4x</td>
</tr>
<tr>
<td>scan</td>
<td>2.78 GiB/s</td>
<td>2.12 GiB/s</td>
<td>8.90 GiB/s</td>
<td>741.14 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>0.8x</td>
<td>3.2x</td>
<td>266.6x</td>
</tr>
<tr>
<td>scatter</td>
<td>1.17 GiB/s</td>
<td>1.12 GiB/s</td>
<td>13.77 GiB/s</td>
<td>914.67 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>1.0x</td>
<td>11.8x</td>
<td>781.8x</td>
</tr>
<tr>
<td>stream</td>
<td>3.94 GiB/s</td>
<td>2.92 GiB/s</td>
<td>24.58 GiB/s</td>
<td>6266.22 GiB/s</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>0.7x</td>
<td>6.2x</td>
<td>1590.4x</td>
</tr>
</tbody>
</table>
ARKOUDA ARGSORT: HERO RUN ON HPE APOLLO SYSTEM WITH IB

- May 2021 hero run performed on large Apollo system
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - used 73,728 cores of AMD Rome
  - ~100 lines of Chapel code
In April 2023, a large HPE Cray EX system with Slingshot-11 set a new record for Arkouda argsort:

- 28 TiB of 8-byte values
- 1200 GiB/s (24 seconds elapsed time)
- used 114,688 cores of AMD Milan
- similar ~100 lines of Chapel code

Not an apples-to-apples comparison

- Different network rates
  - Older one was 100 Gbps IB
  - Newer one was 200 Gbps SS-11
- Different software versions
  - Aggregator optimizations
  - Improvements to the sort: bucket exchange

Arkouda Argsort Performance
• Many of Arkouda’s capabilities also exist in NumPy and Dask
  • Dask implements many NumPy functions to run in distributed memory
  • The “go-to” library for HPC calculations in Python
  • Not necessarily straightforward to program
    – Manual control of tasks / workers

• Small problems done fast – Numpy; Big problems (usually) done fast – Dask

• Problems at any scale done fast – Arkouda

• Some of Arkouda’s most powerful algorithms do not have analogues in Dask (e.g., parallel argsort)

• The following slides show timing comparisons for several key functions
  • Weak scaling (variable node count, variable input size)
  • Chapel 1.27; Dask 2.30.0
DASK VS. ARKOUDA: STREAM TRIAD BENCHMARK

Weak scaling: Time to perform Stream Triad using arrays of size 8 GB per node (median of 50 trials)

\[ C = A + \alpha B \]
DASK VS. ARKOUDA: LOAD HDF5 BENCHMARK

Weak scaling: Time to load 8 GB per node (median of 50 trials)

- Arkouda
- Dask 1 worker/node
- Dask 2 worker/node
- Dask 4 worker/node
- Dask 8 worker/node
- Dask 16 worker/node
- Dask 32 worker/node

Time to completion (seconds) vs. Nodes
DASK VS. ARKOUDA: REDUCE BENCHMARK

Weak scaling: Time to load and sum over 8GB per node (median of 50 trials)

- Arkouda
- Dask 1 worker/node
- Dask 2 worker/node
- Dask 4 worker/node
- Dask 8 worker/node
- Dask 16 worker/node
- Dask 32 worker/node

The diagram shows the time to completion (seconds) as a function of the number of nodes, with Arkouda consistently faster across different worker configurations.
NUMPY VS. ARKOUDA: GATHER BENCHMARK ON UP TO 30 GB DATASETS

\[ C = A[B] \]

![Graph showing performance comparison between Numpy and Arkouda for Gather operation on up to 30 GB datasets. The x-axis represents Array size (GB), the y-axis represents Time to completion (seconds). The graph compares Numpy and Arkouda across different node and thread configurations, highlightingArkouda's faster performance.](image-url)
NUMPY VS. ARKOWDA: GATHER BENCHMARK ON UP TO 2000 GB DATASETS

\[ C = A[B] \]
NUMPY VS. ARKOUDA: SCATTER BENCHMARK ON UP TO 30 GB DATASETS

\[ A[B] = C \]

The graph shows the time to perform Scatter (median of 50 trials) for Numpy and Arkouda with different configurations. It indicates that Arkouda is faster than Numpy for larger datasets.
NUMPY VS. ARKOUDA: SCATTER BENCHMARK ON UP TO 2000 GB DATASETS

\[ A[B] = C \]

Time to perform Scatter (median of 50 trials)

Array size (GB)

Time to completion (seconds)

- Numpy
- Arkouda 1nodes_1threads
- Arkouda 1nodes_6threads
- Arkouda 1nodes_36threads
- Arkouda 2nodes_36threads
- Arkouda 4nodes_36threads
- Arkouda 8nodes_36threads
- Arkouda 16nodes_36threads
- Arkouda 32nodes_36threads
- Arkouda 64nodes_36threads

faster
NUMPY VS. ARKOUDA: ARGSORT ON UP TO 8 GB DATASETS

Time to perform Argsort (median of 50 trials)

- Numpy
- Arkouda 1nodes_1threads
- Arkouda 1nodes_6threads
- Arkouda 1nodes_36threads

Array size (GB)
NUMPY VS. ARKOUDA: ARGSORT ON UP TO 500 GB DATASETS

Time to perform Argsort
(median of 50 trials)

- Numpy
- Arkouda 1nodes_1threads
- Arkouda 1nodes_6threads
- Arkouda 1nodes_36threads
- Arkouda 2nodes_36threads
- Arkouda 4nodes_36threads
- Arkouda 8nodes_36threads
- Arkouda 16nodes_36threads
- Arkouda 32nodes_36threads
- Arkouda 64nodes_36threads

Array size (GB)

Time to completion (seconds)
A Python data analytics framework

- massive scales = dozens of terabytes
- interactive rates = operations that run within the human thought loop (i.e., seconds to small numbers of minutes)
- crucial operations: argsort, gather, scatter, reading from HDF5 and Parquet files
- started with performance and built towards interactivity using a client-server model

High-Performance Highlights

- Great performance and scalability on HPE Apollo and HPE Cray EX
- Faster than Dask at scale
- Outperforms NumPy on a single node

Next Steps

- Enable use in Climate Science by implementing the Python Array API
- Accelerate with GPUs, Josh Milthorpe and others working on at ORNL
- Persistence of data store across and between server sessions

Thank you!

https://github.com/Bears-R-Us/arkouda
https://chapel-lang.org