Hewlett Packard Enterprise

REMOVING TEMPORARY ARRAYS IN ARKOUDA

Ben McDonald, HPE

Software Engineer – Chapel team

CHIUW 2023

June 2, 2023

ARKOUDA SUMMARY

Arkouda

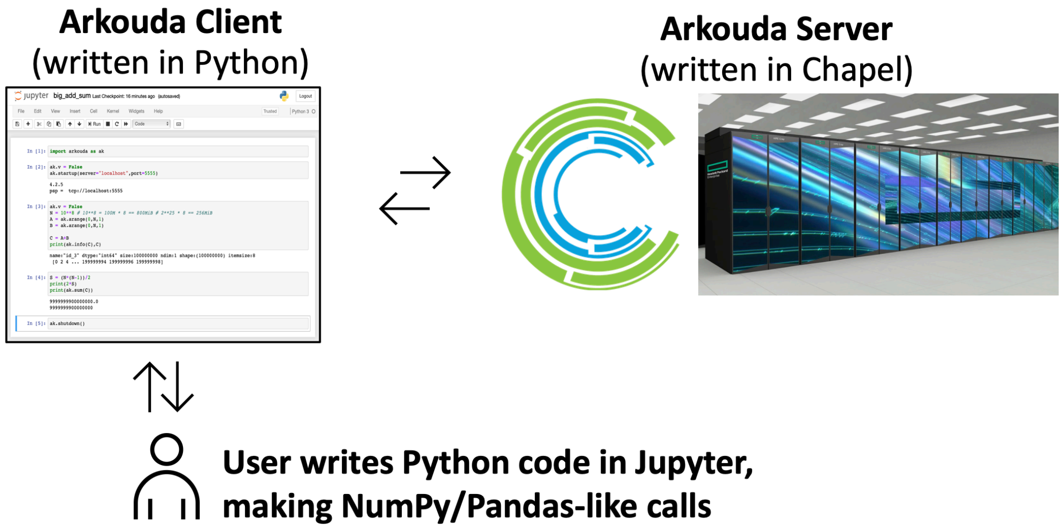
- A Python library supporting a key subset of the NumPy and Pandas interfaces for Data Science
  - Uses a Python-client/Chapel-server model for scalability and performance
  - Computes massive-scale results (multi-TB arrays) within the human thought loop (seconds to a few minutes)
- Open-source: https://github.com/Bears-R-Us/arkouda

Typical Workflow

- Read in hundreds of files containing terabytes of data
- Perform typical data science analysis on the data
  - i.e., binary operations, sort, etc.
- Evaluate results/write to file

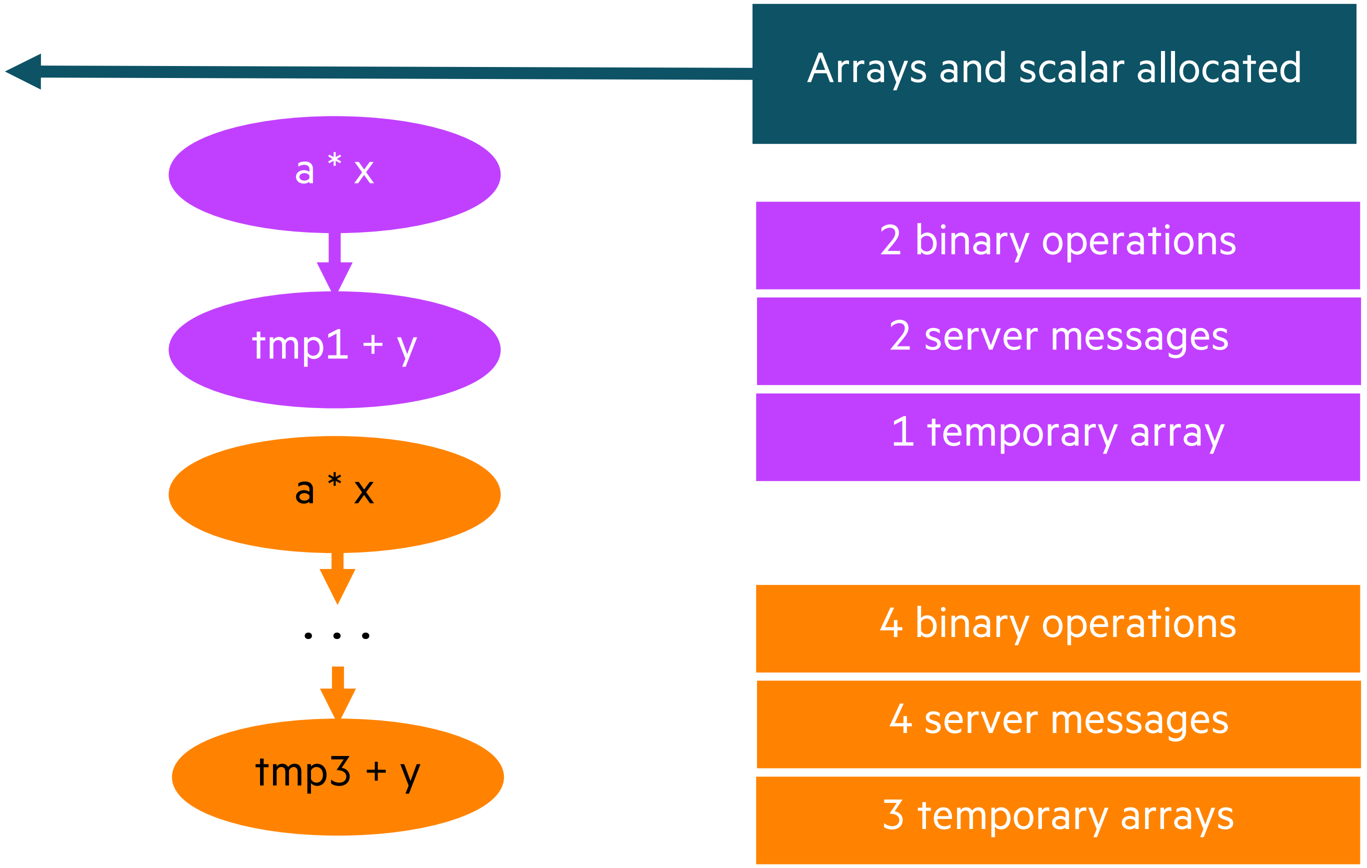
Note

- Commands are sent and executed separately



CURRENT ARKOUDA EVALUATION MODEL

```
>>> a = ak.randint(...)
>>> x = ak.randint(...)
>>> y = 5

...

>>> a * x + y
<result-array>

...

>>> a * x + y * a + y
<result-array>
```



- There are several messages, several temporary arrays created for a single expression

ARKOUDA EVALUATION MODEL

Evaluation Model

- Commands are sent one at a time from the Python client to be evaluated by the Chapel server
  - Human-readable response returned to Python client
- Each command is sent individually, even if written as a single expression (e.g., 'a * x + y')

Ideal for…

- …executing complex, compute-intensive operations requiring only a single message
  - E.g., argsort, group by, etc.

Not ideal for…

- …executing simple expressions or small blocks of code requiring many messages
  - E.g., binary operations, Python code using many different Arkouda functions, etc.

Idea for the best of both worlds…

- Send compound expressions to the server in one message, avoiding multiple messages and temporary arrays
  - This could reduce memory footprint, improve performance of compound expressions, and enable new Arkouda features

ARKOUDA LISP INTERPRETER

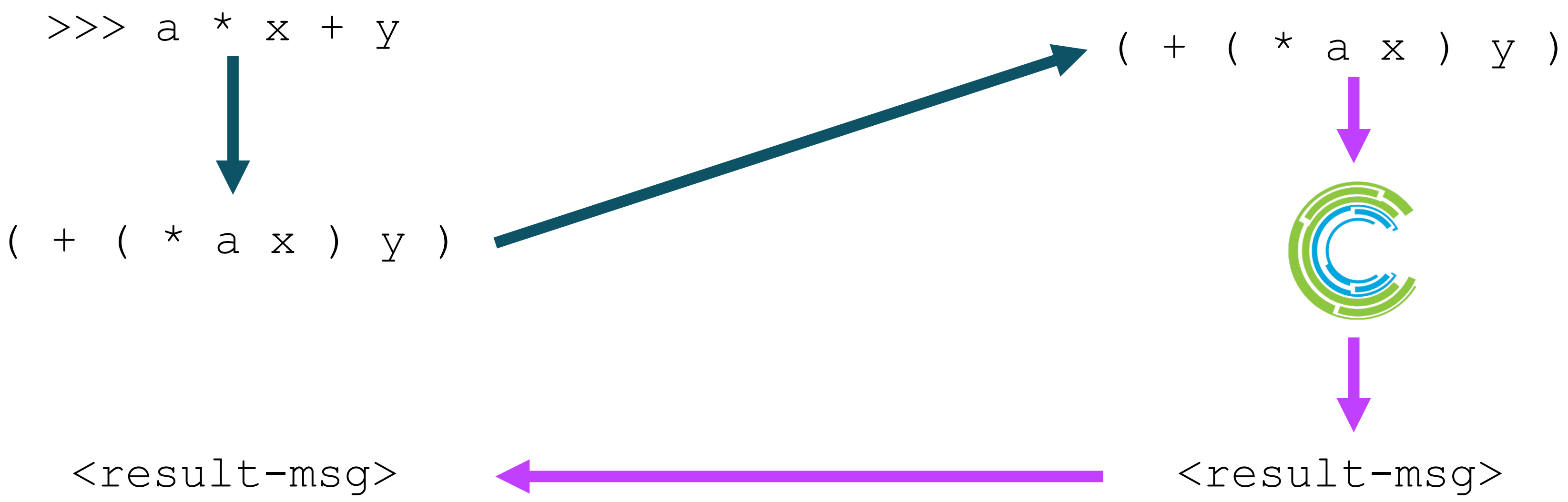
LISP INTERPRETER IMPLEMENTATION

Python client

1. Create AST out of Python expression
2. Convert AST into Lisp expression
3. Send Lisp expression to server

Chapel server

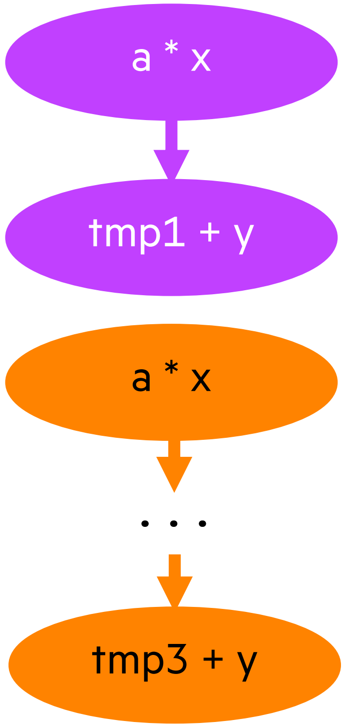
1. Parse Lisp expression
2. Evaluate expression in-place
3. Return result to client



LISP INTERPETER ARKOUDA EVALUATION MODEL

```
>>> a = ak.randint(...)
>>> x = ak.randint(...)
>>> y = 5

...

>>> a * x + y
<result-array>


...

>>> a * x + y * a + y
<result-array>
```

Arrays and scalar allocated



2 binary operations

1 server messages

0 temporary array

4 binary operations

1 server messages

0 temporary arrays

- There are ~~several~~ messages, ~~several~~ temporary arrays created for a single expression
- There is **one** message, **zero** temporary arrays created for each expression

ARKOUDA LISP INTERPRETER

Usage

- Functions defined with the ‘@arkouda_func’ decorator are converted to lisp and sent to server in 1 message
  - Arbitrary Python code can be executed as a single message on the server side if parsing has been implemented

```
@arkouda_func
def my_func(v,x,y):
    (a := v*10)
    return ((y+1) if   (not (x < a))
                  else (y-1))
```

Benefits

- New functionality can be supported (e.g., Arkouda functions shown above)
- Memory footprint reduced in compound expressions by reducing number of temporary arrays
- Communication between client and server requires only a single message
  - As opposed to ‘numOps’ messages with original Arkouda model
- Potential for improved performance, evaluating entire expression at once, rather than piecewise

PERFORMANCE OPTIMIZATIONS & RESULTS

PERFORMANCE RESULTS

Performance results

- Simple 'a * x + y' operation used to gauge performance
  - Worst case for lisp interpreter, since it is only removing 1 temporary/1 message; greater benefit as complexity increases
- Numbers collected on a single node of a Cray CS, elements of type 'real(64)' used for evaluation

Version	1,000,000 Element Throughput	10,000,000 Element Throughput
Arkouda	4.35 GiB/s	31.45 GiB/s
Initial Lisp	0.21 GiB/s	0.29 GiB/s

- Initial performance numbers were over 100x behind standard lisp interpreter

Sources of overhead

- Evaluating entire lisp expression for each element of the array, even though it is identical each time
- Dynamic allocations of class-based data structures used to parse lisp expression
- Casting of expression tokens to concrete types, since types are not known at compile time

PERFORMANCE OPTIMIZATIONS

Two main optimizations

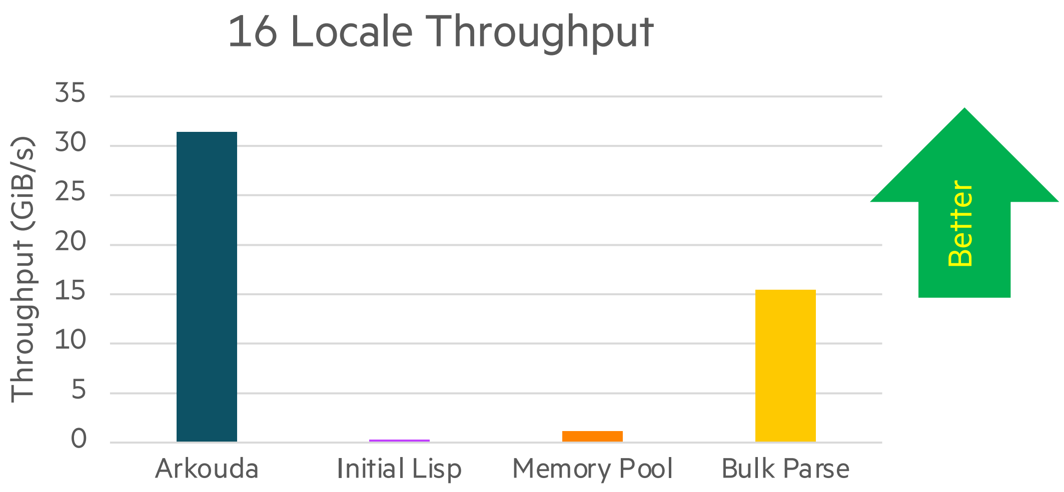
1. Implement a “memory pool” to reduce heap-allocations
   - Each symbol in the lisp expression dynamically allocating a class object when parsing
   - Memory pool optimization returns each allocated object back to a memory pool once finished
   - Heap allocations ~30x more expensive than binary operations, so significant slowdown from each allocation

2. Chunk the array, parsing the lisp expression once for each task, rather than once for each element
   - The original code performed was parsing identical lisp expressions for each element in the array

PERFORMANCE RESULTS

- Throughput to evaluate ‘a * x + y’ using Arkouda arrays on a single node of a Cray CS:
  - Higher is better on performance graph

Version	1,000,000 Element Throughput	10,000,000 Element Throughput
Arkouda	4.35 GiB/s	31.45 GiB/s
Initial Lisp	0.21 GiB/s	0.29 GiB/s
Memory Pool	1.02 GiB/s	1.13 GiB/s
Bulk Parse	3.82 GiB/s	15.45 GiB/s



- Lisp interpreter improved over 50x after optimizations, but still ~2x behind standard Arkouda evaluation model

CONCLUSION

- The lisp interpreter provides new functionality into Arkouda and reduces memory footprint
  - Performance still ~2x behind the standard Arkouda model
- Majority of additional overhead has been identified being spent in casting abstract tokens to values
  - This is required in order to support multiple different types
  - Overhead could be cut out by only supporting a single data type or having datatype-specific implementations
- Through experimentation, theoretical performance ceiling has been shown to be ~2x over base Arkouda

THANK YOU

https://chapel-lang.org
@ChapelLanguage