Improving a High Productivity Data Analytics Chapel Framework

Prashanth Pai (Rice University), Andrej Jakovljević (University of Belgrade), Dr. Zoran Budimlić (Rice University), Dr. Costin Iancu (Lawrence Berkeley National Laboratory)
Introduction

Arkouda
- Cutting edge Python data science library
- Interactive Python client
- Powerful Chapel-backed server

Many message exchanges
- Client and server's form of communication
- Inefficient and costly in some cases

Optimized client architecture
- Supports lazy evaluation
- Caches/reuses arrays and function results
- Eliminates common subexpressions
- Reduces unnecessary message exchanges
Section 1

Arkouda Overview
What is Arkouda?

Software package whose purpose is to optimize data science operations on large distributed data sets using parallel computations and a NumPy-like syntax.

Ties exploratory data analysis with high-performance computing models to complement existing frameworks such as Pandas.

Supports multi-locale processing of linear algebra data sets and graphs.

More information: https://github.com/Bears-R-Us/arkouda
Arkouda Architecture

Python Client
- Exposes simple API to parallel operations
- Communicates with server using single command protocol

Chapel-backed server
- Implements set of data operations
- Receives and replies to messages
- Performs computations in parallel
- Stores data
I need a 10-element array of random integers from 0 to 10

Creating proxy/storing id

Creating/storing array

The array has been created and its id is 1
Pdarray

Specifics

Class which overloads common functions such as addition and multiplication

Relies on the Python compiler for garbage collection (invoking deletion)

Everything gets turned into a message
Arkouda Example

- A = ak.randint(0, 10, 10)
- B = (A * A) + (A * A)
- C = ak.randint(0, 10, 10)
- print(C)
- Sends 8 messages and creates 5 arrays
  - 2 -> randint
  - 3 -> binopvv
  - 2 -> delete
  - 1 -> str
Arkouda Example

- $A = \text{ak.randint}(0,10,10)$
- $B = (A * A) + (A * A)$
- $C = \text{ak.randint}(0,10,10)$
- print(C)
- Sends 8 messages and creates 5 arrays
  - 2 -> randint
  - 3 -> binopvv
  - 2 -> delete
  - 1 -> str
- Sends 8 messages and creates 5 arrays
- $2 > \text{randint}$
- $3 > \text{binopvv}$
- $2 > \text{delete}$
- $1 > \text{str}$
Arkouda Example

- $A = \text{ak.randint}(0, 10, 10)$
- $B = (A \times A) + (A \times A)$
- $C = \text{ak.randint}(0, 10, 10)$
- `print(C)`
- Sends 8 messages and creates 5 arrays
  - 2 -> `randint`
  - 3 -> `binopvv`
  - 2 -> `delete`
  - 1 -> `str`
A = ak.randint(0,10,10)
B = (A * A) + (A * A)
C = ak.randint(0,10,10)
print(C)

Sends 8 messages and creates 5 arrays
- 2 -> randint
- 3 -> binopvv
- 2 -> delete
- 1 -> str
Arkouda Example

- $A = \text{ak.randint}(0,10,10)$
- $B = (A \times A) + (A \times A)$
- $C = \text{ak.randint}(0,10,10)$
- print($C$)
- Sends 8 messages and creates 5 arrays
  - 2 -> randint
  - 3 -> binopvv
  - 2 -> delete
  - 1 -> str
Section 2

Optimized Framework
## Client-Side Optimization

| Reduce                  | Reduce ZeroMQ messages between server and client using batching and code analysis |
|-------------------------|----------------------------------------------------------------------------------|
| Optimize                | Optimize programs using lazy evaluation and common subexpression elimination     |
| Reuse                   | Reuse unused temporary arrays and improve server-side memory management          |
Room for Improvement

- $A = \text{ak.randint}(0, 10, 10)$
- $B = (A * A) + (A * A)$
- $C = \text{ak.randint}(0, 10, 10)$
- `print(C)`
- Sends 2 messages and creates 1 array
  - 1 -> `randint`
  - 1 -> `str`
Room for Improvement

- $A = \text{ak.randint}(0,10,10)$
- $B = (A \times A) + (A \times A)$
- $C = \text{ak.randint}(0,10,10)$
- print(C)
- Sends 2 messages and creates 1 array
  - 1 -> randint
  - 1 -> str

Eliminated a common subexpression
Room for Improvement

- $A = \text{ak.randint}(0, 10, 10)$
- $B = (A \times A) + (A \times A)$
- $C = \text{ak.randint}(0, 10, 10)$
- $\text{print}(C)$
- Sends 2 messages and creates 1 array
  - 1 -> randint
  - 1 -> str
- Reused a server-side array
Room for Improvement

- A = ak.randint(0, 10, 10)
- B = (A * A) + (A * A)
- C = ak.randint(0, 10, 10)
- print(C)
- Sends 2 messages and creates 1 array
  - 1 -> randint
  - 1 -> str
Architecture Components

Command Buffer
- Enables lazy evaluation and CSE which reduces sent messages and server-side arrays

Cache
- Allows reuse of server-side arrays which improves space efficiency

Store Functions
- Extends server API to overwrite arrays
Batch Command Buffer Data Structure

- FIFO queue of arkouda commands (BufItems)
- Hold a reference to a BufItem in each pdarray
- BufItem properties
  - Write id string
  - Comma separated read id string
  - Command string
  - Comma separated arg string
- Lazy evaluation on data access demand

```
A = ak.randint(0,10,10)
B = ak.randint(0,10,10)
C = A+B
```
Caching: Optimize Server-Side Memory Management

- Minimize messages that create/destroy server-side arrays
- Keep track of server-side arrays and manage them on the client side
- Map client array ids to server array ids
- Reduce work done by server (creation/destruction of arrays)

```python
for x in range(1000):
    A = ak.randint(0,10,10)
    B = ak.randint(0,10,10)
    C = A + B
    print(C)
```

Only use 3 arrays instead of 3000
Server API Extension: Store Functions

Specify where to store the result of arkouda operations

Allows us to use cached unused server arrays
Putting it all together

B = (A * A) + (A * A)
print(B)
\[ B = (A \times A) + (A \times A) \]

print(B)

Putting it all together
Putting it all together

B = (A * A) + (A * A)
print(B)
Putting it all together

\[ B = (A \times A) + (A \times A) \]

\[
\text{print}(B)
\]
Putting it all together

\[ B = (A \times A) + (A \times A) \]

\[ \text{print}(B) \]
*All experiments were run on single shared memory node with a Xeon E3-1220 [7] processor
Example: Triangle Count for Dense Matrices

```python
maxi = 0
arr = np.zeros(len(A), np.int64)
for i in range(len(A)):
    for j in range(len(A)):
        k = ak.sum(A[i] * A.transpose()[j])
        arr[j] = k
        pdarr = ak.array(arr)
    maxi += ak.sum(pdarr * A[i])
return maxi
```

Simulates matrix – matrix multiplication and applies a mask sum((L * L) .*L)

Opportunities to reuse arrays
Ratio of Created Arrays between Base/Optimized Arkouda (Dense)

- Fewer arrays used in optimized version across the board
- Scales based on size of matrix
- Nearly 30,000 times fewer created arrays on largest example
Profiling of Base/Optimized Arkouda (Dense)

- Faster program execution across the board
- Significantly reduced time creating/deleting arrays on Chapel
for i in range(len(pointers) - 1):
    right = pointers[i + 1]
    if (pointers[i] < right):
        for j in range(pointers[i], right):
            s += ak.sortIntersect1d(find_splice(i, pointers, pd_indexes),
                                      find_splice(pd_indexes[j], pointers2, pd_indexes2)).size

• Uses CSR, CSE and Arkouda set operations
• Also room for array reuse
Performance Improvements as a Percentage (Sparse)

- Performance improvements across the board, especially for more sparse matrices
  - Less time spent on set operations
- Over 120% performance improvement for delanay_n15
Example: Betweenness Centrality

- Measures the number of shortest paths that go through a node, divided by the number of shortest paths in the graph
- Mirrored off of a GraphBLAS implementation
- Two loops that each generate a series of temporaries
- $O(n^2)$ rather than $O(n^3)$ so easier to use larger matrices
Profiling of Betweenness Centrality

- Consistent performance improvements across the board
- Improvements not dependent on matrix size since Arkouda array size was consistent
Example: NYC Taxi Cab Example

- Real-world example based on database of NYC taxi trips in January 2020
- Series of unary operations applied to a single immutable Arkouda array
- Repetition of `min()`, `max()`, `mean()`, `std()`
  - Internal repetition of `sum()`
- Results
  - 35% performance increase (0.16s rather than 0.25s)
  - Sent 30 messages rather than 36 messages
Section 4

Current Work & Next Steps
Current Work: Message Aggregation in Loops

- Example on left has 3 calls per loop iteration
  - (3 * number of iterations) message exchanges
- Can we generalize these loops on the server-side?
- Key idea: send entire loops from the client to the server using one message

```python
for i in range(len(array)):
    b = array[i]
    b = b - m
    b = b // e
    b = b % r
    b = math.floor(b)
    c = buckets[b]
    c = c + 1
    buckets[b] = c
```
# Preliminary Results

3Arkouda operations, 4 scalar operations

| Iterations | Base (s) | Dependent (s) | Base-Dependent Ratio | Independent (s) using (iterations/100) tasks | Base-Independent Ratio |
|------------|----------|---------------|----------------------|---------------------------------------------|------------------------|
| 1k         | 3.7261   | 0.0769        | 48.4311              | 0.0760                                      | 49.0476                |
| 10k        | 40.3329  | 0.7139        | 56.4969              | 0.7046                                      | 57.2431                |
| 100k       | 379.6306 | 6.9259        | 54.8129              | 7.4018                                      | 51.2891                |
| 1M         | 3223.7308| 67.1780       | 47.9879              | 82.4395                                     | 39.1042                |

* Server/client run on MacBookPro, single node, 2.4 GHz Quad-Core Intel Core i5, shared memory
# Preliminary Results

13 Arkouda operations, 4 scalar operations

| Iterations | Base (s) | Dependent (s) | Base-Dependent Ratio | Independent (s) using (iterations/100) tasks | Base-Independent Ratio |
|------------|----------|---------------|-----------------------|---------------------------------------------|------------------------|
| 1k         | 15.6984  | 0.2271        | 69.1344               | 0.2401                                      | 65.3837                |
| 10k        | 164.7580 | 2.3256        | 70.8459               | 2.1745                                      | 75.7677                |

* Server/client run on MacBookPro, single node, 2.4 GHz Quad-Core Intel Core i5, shared memory
Looking Ahead

- Support larger subset of Arkouda operations
- Support more complex benchmarks
  - Nested loops and arrays as variables (dense triangle count), helper functions (sparse triangle count)
- Create easier-to-use client-side API
- Run server on distributed system to fine-tune number of tasks
Key Takeaways

Arkouda
- Powerful, interactive framework
- Room for improvement (memory footprint and messages)

Our architecture
- Implements lazy evaluation, caching, and CSE
- Performance increases across the board

Message aggregation
- Another powerful optimization tool
- Significantly reduce message exchanges
Current Milestones

Revised architecture opens opportunities for CSE, lazy evaluation, and space efficiency

Next steps

Polish message aggregation optimization
Combine all optimizations
Move code into Arkouda repo