



Too Big to Fail: **Massive Scale Linear Algebra** **with Chapel and Arkouda**

Chris Hollis

Software Engineer – Department of Defense

CHI UW 2023

June 2

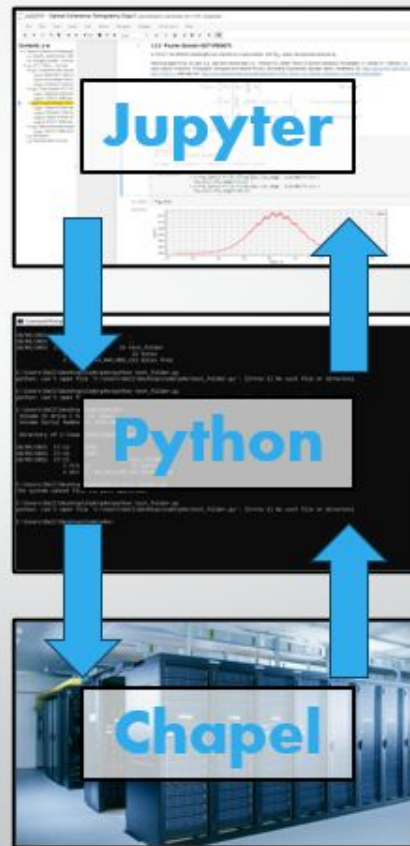
Objective

- Exploratory data analysis (EDA) requires open-ended and frictionless interaction with data
 - Pandas -> NumPy/SciPy -> Linear algebra -> Pandas
- Arkouda allows interactive EDA at scale
 - 10's of TBs of data
 - Distributed memory allows for large array allocation



Arkouda Overview

- What is Arkouda?
 - A NumPy-like Python app that utilizes Chapel for its backend server
 - Abstracts powerful Chapel functions with a familiar Python interface
 - Prioritizes compatibility with existing data science workloads
 - Jupyter notebooks
 - Mirrors Pandas/NumPy usage
 - Open-source and can be found at:
 - <https://github.com/Bears-R-Us/arkouda>



AkSparse Overview

- What is AkSparse?
 - Sparse linear algebra library built with Arkouda
 - Emulates SciPy's ".sparse" library
 - Supports COO, CSR, and CSC formats
 - Basic matrix arithmetic
 - Matrix-Vector multiplication
 - Sparse General Matrix Multiplication (SpGeMM)

	Sparse matrix object	Format conversion	Sparse General Matrix Multiplication
Aksparse	<code>Aksparse.coo_matrix()</code>	<code>A.tocsc()</code>	<code>C = A.spgemm(B)</code>
scipy	<code>scipy.coo_matrix()</code>	<code>A.tocsr()</code>	<code>C = A.dot(B)</code>

Algorithm

- **Sparse matrix multiplication is hard**
 - No way to know how large solution will be beforehand
 - Load balancing
 - Communication cost
- **Focus on large unstructured data**
 - Need distributed-scale computing
 - Communication cost is a bottleneck

$$\begin{pmatrix} A \end{pmatrix} \begin{pmatrix} B \end{pmatrix} = \begin{pmatrix} ? \end{pmatrix}$$

Algorithm

- **Sparse matrix multiplication is hard**
 - No way to know how large solution will be beforehand
 - Load balancing
 - Communication cost
- **Focus on large unstructured data**
 - Need distributed-scale computing
 - Communication cost is a bottleneck
- **How is AkSparse's SpGeMM different?**
 - Leverage Arkouda's optimized sorting and groupby capabilities on HPC hardware
 - Interactive manipulation of TB scale data
 - "Outer product" formulation of SpGeMM
 - Reveals size of work needed before any computations
 - Minimize communication cost through Arkouda's message aggregation

Algorithm

$$\begin{pmatrix} A \end{pmatrix} \begin{pmatrix} B \end{pmatrix} = \begin{pmatrix} C \end{pmatrix}$$

Algorithm

$$i \begin{pmatrix} \text{---} \end{pmatrix} \begin{pmatrix} | \end{pmatrix} = \begin{pmatrix} \square_{ij} \end{pmatrix}$$

The diagram illustrates the dot product of a row vector and a column vector. The first vector is a row vector with a green horizontal bar, labeled 'i' on the left. The second vector is a column vector with a green vertical bar, labeled 'j' below it. An equals sign follows, leading to a single green square element labeled 'i,j' below it, representing the result of the dot product.

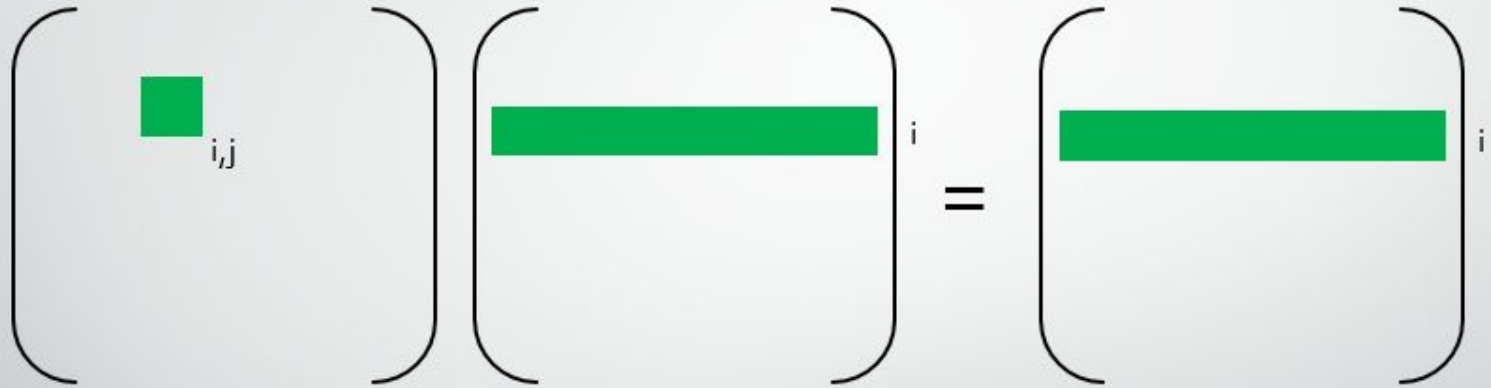
Algorithm

$$\begin{pmatrix} \blacksquare_{ij} & & \\ & & \\ & & \end{pmatrix} \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}$$

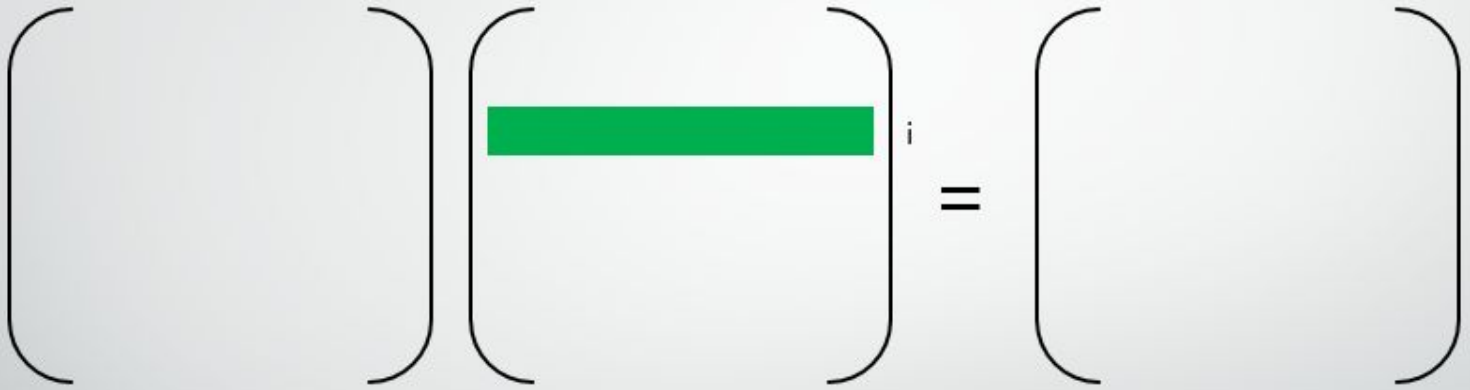
Algorithm

$$\begin{pmatrix} \blacksquare_{ij} \\ \\ \\ \end{pmatrix} \begin{pmatrix} \\ \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \\ \\ \end{pmatrix}$$

Algorithm

$$\begin{pmatrix} \blacksquare_{i,j} \end{pmatrix} \begin{pmatrix} \text{---} \end{pmatrix}_i = \begin{pmatrix} \text{---} \end{pmatrix}_i$$
The diagram shows three matrices in large parentheses. The first matrix contains a single green square labeled 'i,j' at its center. The second matrix contains a green horizontal bar representing a row vector, with the label 'i' to its right. An equals sign follows. The third matrix contains a green horizontal bar representing a row vector, with the label 'i' to its right.

Algorithm

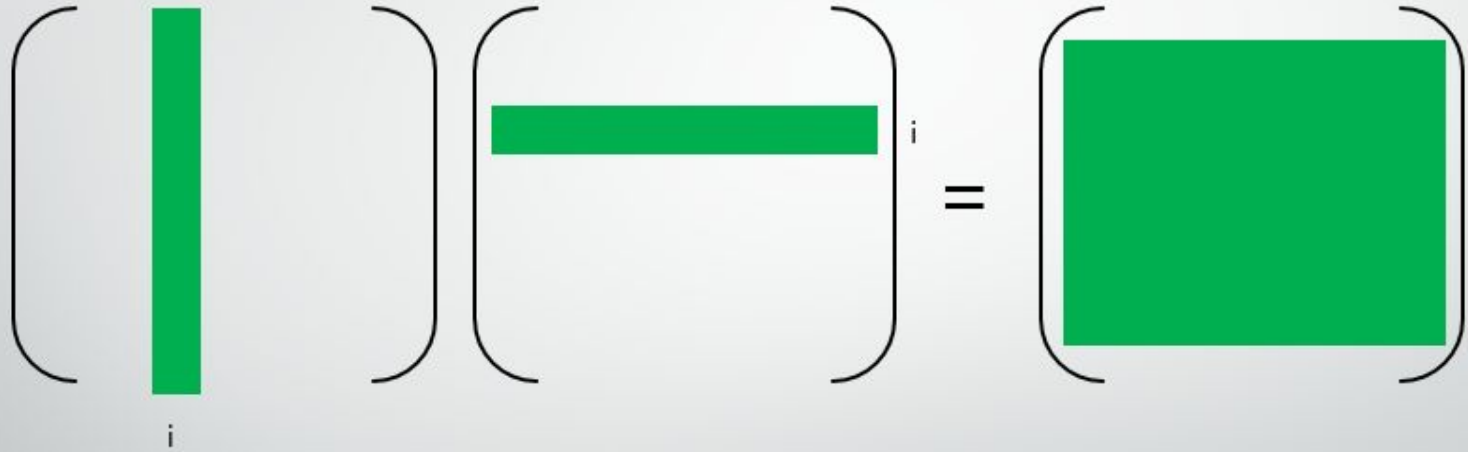

$$\begin{pmatrix} \\ \\ \end{pmatrix} \begin{pmatrix} \\ \\ \end{pmatrix} = \begin{pmatrix} \\ \\ \end{pmatrix}$$

Algorithm

$$\begin{pmatrix} \color{green}{|} \\ \color{green}{|} \\ \color{green}{|} \\ \color{green}{|} \\ \color{green}{|} \end{pmatrix} \begin{pmatrix} \color{green}{-} \\ \color{green}{-} \\ \color{green}{-} \\ \color{green}{-} \\ \color{green}{-} \end{pmatrix} = \begin{pmatrix} \phantom{\color{green}{|}} \\ \phantom{\color{green}{|}} \\ \phantom{\color{green}{|}} \\ \phantom{\color{green}{|}} \\ \phantom{\color{green}{|}} \end{pmatrix}$$

The diagram illustrates the multiplication of a column vector and a row vector. The first vector is a column vector with five green vertical bars, and the second is a row vector with five green horizontal bars. The result is an empty vector with five slots, representing a zero vector.

Algorithm



Algorithm

- SpGeMM in Aksparse (A.spgemm(B))
 - Convert A to CSC
 - Convert B to CSR
 - Find all 'hits' between nonzero entries in the columns of A and corresponding rows of B
 - Generate a single Arkouda array for all the multiplications of A.spgemm(B)
 - Perform a GroupBy on the matrix indices implied by the full multiplication array
 - Perform a sum aggregate on the full multiplication array results to yield the final matrix C

```
def spgemm(self: CSC, other: CSR):
```

```
#Identify number of multiplications needed
```

```
starts = other.indptr[self._gb_col_row.unique_keys[0]]
ends = other.indptr[self._gb_col_row.unique_keys[0] + 1]
lengths = (ends - starts)
fullsize = lengths.sum()
segs = ak.cumsum(lengths) - lengths
slices = ak.ones(fullsize, dtype=ak.akint64)
diffs = ak.concatenate((ak.array([starts[0]]),
                          starts[1:] - ends[:-1] + 1))
```

```
#Set up arrays for multiplication
```

```
slices[segs] = diffs
nonzero = (ends > starts)
fullsegs, ranges = segs, ak.cumsum(slices)
fullBdom = other._gb_row_col.unique_keys[1][ranges]
fullAdom = ak.broadcast(fullsegs,
                          self._gb_col_row.unique_keys[1][nonzero],
                          fullsize)
fullBval = other.data[ranges]
fullAval = ak.broadcast(fullsegs, self.data[nonzero], fullsize)
fullprod = fullAval * fullBval
```

```
#GroupBy indices and perform aggregate sum
```

```
proddomGB = ak.GroupBy([fullAdom, fullBdom])
result = proddomGB.sum(fullprod)
return Csr(result[1],
            result[0][1],
            result[0][0],
            shape = (self.shape[0], other.shape[1]))
```

Algorithm

- SpGeMM in Aksparse [A.spgemm(B)]

- Convert A to CSC

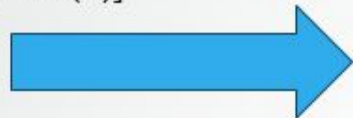
- Convert B to CSR

- Find all 'hits' between nonzero entries in the columns of A and corresponding rows of B

- Generate a single Arkouda array for all the multiplications of A.spgemm(B)

- Perform a GroupBy on the matrix indices implied by the full multiplication array

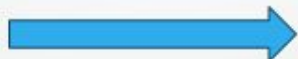
- Perform an aggregate on the full multiplication array results to yield the final matrix C



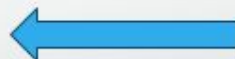
Why it's an outer product



This is easy!

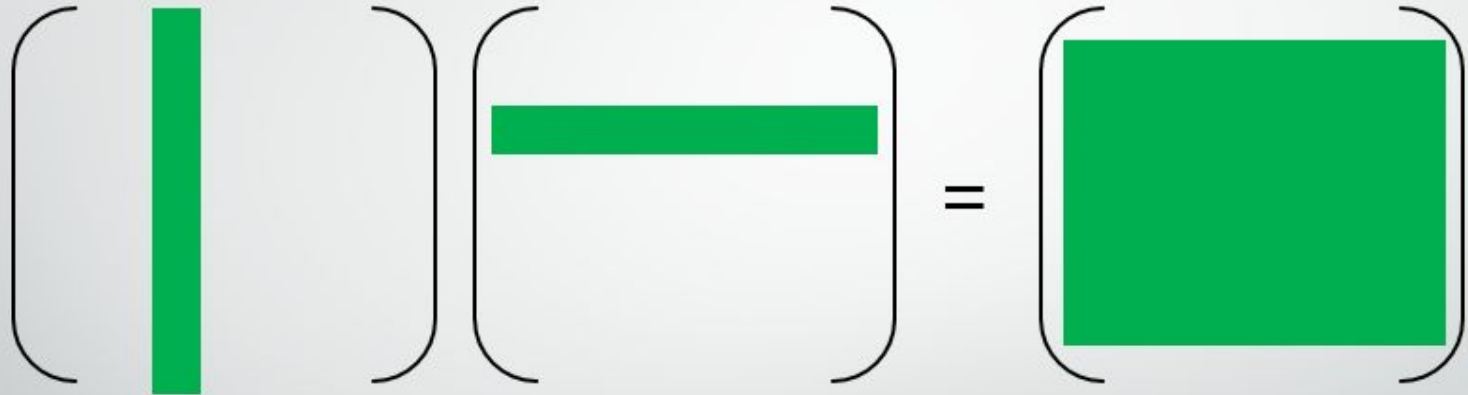


Effectively yields N rank 1 COO matrices



This is hard!

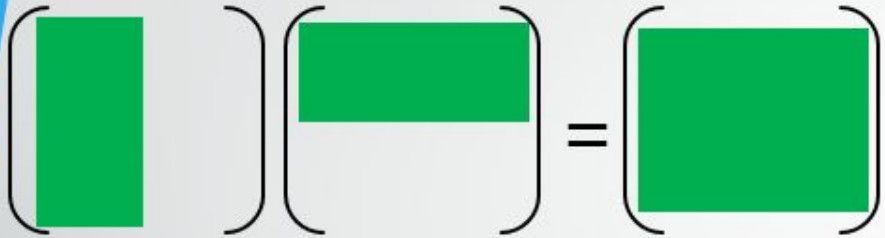
Algorithm



Algorithm



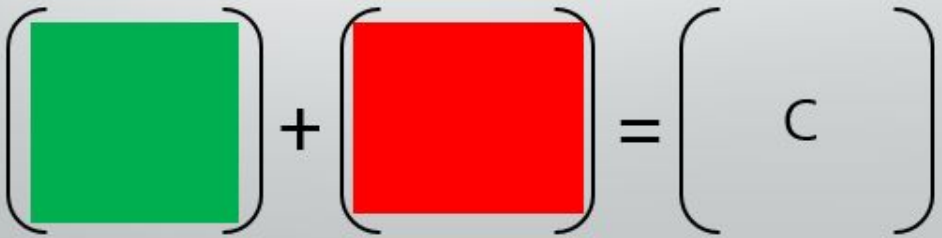
Algorithm



A diagram illustrating matrix multiplication. On the left, a vertical green rectangle is enclosed in a pair of parentheses, followed by an equals sign, then a horizontal green rectangle also enclosed in a pair of parentheses. To the right of this is a square green rectangle enclosed in a pair of parentheses.



A diagram illustrating matrix multiplication. On the left, a vertical red rectangle is enclosed in a pair of parentheses, followed by an equals sign, then a horizontal red rectangle also enclosed in a pair of parentheses. To the right of this is a square red rectangle enclosed in a pair of parentheses.



A diagram illustrating matrix addition. On the left, a square green rectangle is enclosed in a pair of parentheses, followed by a plus sign, then a square red rectangle also enclosed in a pair of parentheses. To the right of this is a large pair of parentheses containing the letter 'c'.

Algorithm

- Only "difficult" computation: $O(\# \text{ mults})$ groupby
 - $O(\# \text{ mults})$ in general is much bigger than nnz
 - Arkouda is tuned to handle large sorts
- Counting # of mults needed is easy
 - Can be done without forming the array
 - This means we know if splitting the problem is necessary before attempting the calc
- Avoids the load balancing issue
 - Recursive splitting
- Runs on large distributed memory system
 - Multiple 2TB nodes with dual-socket InfiniBand interconnect
 - Can handle MUCH larger nnz amounts in the output

Results/Benchmarks

SciPy on home computer

NNZ/Size(NxN)	100k	1mil	10mil	100mil
100k	0.02	0.04	0.14	1.02
1mil	0.34	0.35	0.46	2.01
10mil	311.14	11.64	6.28	4.97
100mil	X	X	X	X
1bil	X	X	X	X

SciPy on HPC

NNZ/Size(NxN)	100k	1mil	10mil	100mil
100k	0.02	0.04	0.24	2.52
1mil	0.28	0.23	0.53	3.07
10mil	24.57	6.59	4.56	6.84
100mil	X	X	101.50	55.68
1bil	X	X	X	X

Arkouda on HPC

NNZ/Size(NxN)	100k	1mil	10mil	100mil
100k	3.19	3.10	3.03	2.98
1mil	3.22	3.18	3.19	3.11
10mil	7.51	3.46	3.22	3.14
100mil	41.79	42.01	7.25	3.50
1bil	X	X	X	44.99

*Results in seconds

Final Test case:

~10bil nnz in output
~10bil multiplications

Results/Benchmarks

- Adjacency matrix A for a "small world" graph
 - #rows = #columns = ~77 mil
 - NNZ = ~620 mil
- Computing $A \cdot A^T$
 - 40 compute nodes
 - ~440 billion multiplies
 - ~300 billion NNZ in C
 - ~22TB of memory to compute and store solution
 - ~4 mins



Future Work

- AkSparse is open source and available here:
 - <https://github.com/Bears-R-Us/arkouda-contrib/tree/main/aksparse>
- Additional linear algebra functionality
- Optimization
 - Improved load balancing
 - Implement outerproduct SpGemm Chapel kernel
- Problems too big to store in memory (write to disk)
 - Target "big" problem:
 - #edges = $O(1 \text{ bil})$
 - NNZ = (100 bil)
 - # multiplies = $O(100 \text{ trillion})$



Questions?