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

Chris Hollis
Software Engineer – Department of Defense
CHIUW 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)

<table>
<thead>
<tr>
<th>AkSparse</th>
<th>Sparse matrix object</th>
<th>Format conversion</th>
<th>Sparse General Matrix Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Aksparse.coo_matrix()</td>
<td>A.tocsc()</td>
<td>C = A.spgemm(B)</td>
</tr>
<tr>
<td>scipy</td>
<td>scipy.coo_matrix()</td>
<td>A.tocsr()</td>
<td>C = A.dot(B)</td>
</tr>
</tbody>
</table>
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
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

\[ A \times B = C \]
Algorithm

\[ i \quad \mid \quad j \quad = \quad i_{ij} \]
Algorithm
Algorithm

\[
\begin{align*}
\text{Algorithm} & : & \\
\begin{pmatrix}
i & j
\end{pmatrix} & \begin{pmatrix}
i
\end{pmatrix} & = & \begin{pmatrix}
i
\end{pmatrix}
\end{align*}
\]
Algorithm

\[ H_i = \]
Algorithm

\[
\begin{bmatrix}
  H
\end{bmatrix}
\begin{bmatrix}
  i
\end{bmatrix}
= 
\begin{bmatrix}
  \mathbf{H}i
\end{bmatrix}
\]
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

```python
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.int64)
    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

\[
\begin{align*}
\left[ \begin{array}{c}
\text{Green} \\
\text{Red}
\end{array} \right] & \left[ \begin{array}{c}
\text{Green} \\
\text{Red}
\end{array} \right] = \\
\left[ \begin{array}{c}
\text{Green} \\
\text{Red}
\end{array} \right] & \left[ \begin{array}{c}
\text{Green} \\
\text{Red}
\end{array} \right] = \\
\left[ \begin{array}{c}
\text{Green} \\
\text{Red}
\end{array} \right] + \\
\left[ \begin{array}{c}
\text{Green} \\
\text{Red}
\end{array} \right] = C
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>NNZ/Size (NxN)</th>
<th>100k</th>
<th>1mil</th>
<th>10mil</th>
<th>100mil</th>
</tr>
</thead>
<tbody>
<tr>
<td>100k</td>
<td>0.02</td>
<td>0.04</td>
<td>0.14</td>
<td>1.02</td>
</tr>
<tr>
<td>1mil</td>
<td>0.34</td>
<td>0.35</td>
<td>0.46</td>
<td>2.01</td>
</tr>
<tr>
<td>10mil</td>
<td><strong>311.14</strong></td>
<td><strong>11.64</strong></td>
<td><strong>6.28</strong></td>
<td><strong>4.97</strong></td>
</tr>
<tr>
<td>100mil</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1bil</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NNZ/Size (NxN)</th>
<th>100k</th>
<th>1mil</th>
<th>10mil</th>
<th>100mil</th>
</tr>
</thead>
<tbody>
<tr>
<td>100k</td>
<td>0.02</td>
<td>0.04</td>
<td>0.24</td>
<td>2.52</td>
</tr>
<tr>
<td>1mil</td>
<td>0.28</td>
<td>0.23</td>
<td>0.53</td>
<td>3.07</td>
</tr>
<tr>
<td>10mil</td>
<td><strong>24.57</strong></td>
<td><strong>6.59</strong></td>
<td><strong>4.56</strong></td>
<td><strong>6.84</strong></td>
</tr>
<tr>
<td>100mil</td>
<td>X</td>
<td>X</td>
<td><strong>101.50</strong></td>
<td><strong>55.68</strong></td>
</tr>
<tr>
<td>1bil</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NNZ/Size (NxN)</th>
<th>100k</th>
<th>1mil</th>
<th>10mil</th>
<th>100mil</th>
</tr>
</thead>
<tbody>
<tr>
<td>100k</td>
<td>3.19</td>
<td>3.10</td>
<td>3.03</td>
<td>2.98</td>
</tr>
<tr>
<td>1mil</td>
<td>3.22</td>
<td>3.18</td>
<td>3.49</td>
<td>3.11</td>
</tr>
<tr>
<td>10mil</td>
<td><strong>7.51</strong></td>
<td><strong>3.46</strong></td>
<td><strong>3.22</strong></td>
<td><strong>3.14</strong></td>
</tr>
<tr>
<td>100mil</td>
<td><strong>41.79</strong></td>
<td><strong>42.01</strong></td>
<td><strong>7.25</strong></td>
<td><strong>3.59</strong></td>
</tr>
<tr>
<td>1bil</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

*Results in seconds

**SciPy on home computer**

**SciPy on HPC**

**Arkouda on HPC**

**Final Test case:**
- ~10bil nnz in output
- ~10bil multiplications
Results/Benchmarks

- Adjacency matrix $A$ for a "small world" graph
  - $\#rows = \#columns \approx 77$ mil
  - $\text{NNZ} \approx 620$ mil
- Computing $AA^T$
  - 40 compute nodes
  - $\approx 440$ billion multiplies
  - $\approx 300$ billion NNZ in C
  - $\approx 22$TB of memory to compute and store solution
- $\approx 4$ 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 bil)
    - NNZ = (100 bil)
    - \# multiplies = O(100 trillion)
Questions?