Tall-and-skinny Matrix Computations in MapReduce

Austin Benson

Institute for Computational and Mathematical Engineering
Stanford University

2nd ICME MapReduce Workshop

April 29, 2013
Collaborators

James Demmel, UC-Berkeley

David Gleich, Purdue

Paul Constantine, Stanford

Thanks!
Matrices and MapReduce

Ax

\| \cdot \|$

A^T A \text{ and } B^T A$

QR and SVD

Conclusion
Two functions that operate on key value pairs:

\[
\begin{align*}
(key, value) & \xrightarrow{\text{map}} (key, value) \\
(key, \langle value_1, \ldots, value_n \rangle) & \xrightarrow{\text{reduce}} (key, value)
\end{align*}
\]

A *shuffle* stage runs between map and reduce to sort the values by key.
MapReduce overview

Scalability: many map tasks and many reduce tasks are used

https://developers.google.com/appengine/docs/python/images/mapreduce_mapshuffle.png
The idea is **data-local** computations. The programmer implements:

- map(key, value)
- reduce(key, \(\langle value_1, \ldots, value_n \rangle\))

The shuffle and data I/O is implemented by the MapReduce framework, e.g., Hadoop.

This is a very **restrictive** programming environment! We sacrifice program control for structure, scalability, fault tolerance, etc.
In MapReduce, we cannot control:

- the number of mappers
- which key-value pairs from our data get sent to which mappers

In MapReduce, we can control:

- the number of reducers
We have matrices, so what are the key-value pairs? The key may just be a row identifier:

\[
A = \begin{bmatrix}
1.0 & 0.0 \\
2.4 & 3.7 \\
0.8 & 4.2 \\
9.0 & 9.0
\end{bmatrix} \rightarrow \begin{bmatrix}
(1, [1.0, 0.0]) \\
(2, [2.4, 3.7]) \\
(3, [0.8, 4.2]) \\
(4, [9.0, 9.0])
\end{bmatrix}
\]

(key, value) → (row index, row)
Matrix representation

Maybe the data is a set of samples

\[
A = \begin{bmatrix}
1.0 & 0.0 \\
2.4 & 3.7 \\
0.8 & 4.2 \\
9.0 & 9.0
\end{bmatrix} \rightarrow \begin{bmatrix}
(“Apr 26 04:18:49”, [1.0, 0.0]) \\
(“Apr 26 04:18:52”, [2.4, 3.7]) \\
(“Apr 26 04:19:12”, [0.8, 4.2]) \\
(“Apr 26 04:22:33”, [9.0, 9.0])
\end{bmatrix}
\]

(key, value) \rightarrow (timestamp, sample)
Matrix representation: an example

Scientific example: \((x, y, z)\) coordinates and model number:

\[
((47570, 103.429767811242, 0, -16.525510963787, \text{iDV7}), [0.00019924, -4.706066e-05, 2.875293979e-05, 2.456653e-05, -8.436627e-06, -1.508808e-05, 3.731976e-06, -1.048795e-05, 5.229153e-06, 6.323812e-06])
\]

**Figure**: Aircraft simulation data. Paul Constantine, Stanford University
What are tall-and-skinny matrices?

A is $m \times n$ and $m \gg n$. Examples: rows are data samples; blocks of $A$ are images from a video; Krylov subspaces
Matrices and MapReduce

$Ax$

$|| \cdot ||$

$A^TA$ and $B^TA$

$QR$ and $SVD$

Conclusion
Slightly more rigorous definition:
It is “cheap” to pass $O(n^2)$ data to all processors.
Ax: Local to Distributed
A may be stored in an uneven, distributed fashion. The MapReduce framework provides load balance.
Ax: MapReduce perspective

The programmer’s perspective for map():

A? → map \( (A?)_i \) \( X \) → Output data
# x is available locally

def map(key, val):
    yield (key, val * x)
We didn’t even need reduce!

The output is stored in distributed fashion:
Matrices and MapReduce

Ax

|| · ||

$A^T A$ and $B^T A$

QR and SVD

Conclusion
Global information $\rightarrow$ need reduce

Examples: $\|Ax\|_1$, $\|Ax\|_2$, $\|Ax\|_\infty$, $|Ax|_0$
Assume we have already computed $y = Ax$. 

Diagram:

- \( y_1 \) and \( y_2 \) 
- \( y_3 \) and \( y_4 \) 
- \( y_5 \) and \( y_6 \)
What can we do with a partial partition of $y$?
We could just compute the squares of each

```python
def map(key, val):
    yield (0, val * val)
```

... then we need to sum the squares
Only one key → everything sent to a single reducer.

```python
def map(key, val):
    # only one key
    yield (0, val * val)

def reduce(key, vals):
    yield ('norm2', sum(vals))
```
How can this be improved?

```python
1 def map(key, val):
2     # only one key
3     yield (0, val * val)
4
5 def reduce(key, vals):
6     yield ('norm2', sum(vals))
```
Idea: Use more reducers

```python
def map1(key, val):
    key = uniform_random([1, 2, 3, 4, 5, 6])
    yield (key, val * val)

def reduce1(key, vals):
    yield (key, sum(vals))

def map2(key, val):
    yield key, val

def reduce2(key, vals):
    yield ('norm2', sum(vals))

map1() → reduce1() → map2() → reduce2()
$\|y\|_2^2$ improvement

\[
\begin{array}{c|c}
y_1 & y_2 \\
\hline
y_3 & y_4 \\
\hline
y_5 & y_6 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{y}^T_1y_1 & \text{y}^T_2y_2 \\
\hline
\text{y}^T_3y_3 & \text{y}^T_4y_4 \\
\hline
\text{y}^T_5y_5 & \text{y}^T_6y_6 \\
\end{array}
\]

\[
\text{y}^T_1y_1 + \ldots + \text{y}^T_6y_6
\]
Problem: $O(m)$ data emitted from mappers in first stage.

Problem: 2 iterations.

Idea: Do partial summations in the map stage.
This is the idea of a combiner.

$O(\#(\text{mappers}))$ data emitted from mappers.
Suppose we only care about $\|Ax\|_2^2$, not $y = Ax$ and $\|y\|_2^2$.

Can we do better than:

1. compute $y = Ax$
2. compute $\|y\|_2^2$

Of course!
Combine our previous ideas:

```python
1  def map(key, val):
2      yield (0, (val * x) * (val * x))
3
4  def reduce(key, vals):
5      yield sum(vals)
```
We can easily extend these ideas to other norms

Basic idea for computing $\|y\|$:

1. perform some independent operation on each $y_i$
2. combine the results
\[ \| Ax \|_0 \]

\[
\begin{align*}
\text{def } & \text{map_abs(key, val):} \\
& \quad \text{yield } (0, \mid value \ast x \mid) \\
\text{def } & \text{map_square(key, val):} \\
& \quad \text{yield } (0, (value \ast x)^2) \\
\text{def } & \text{map_zero(key, val):} \\
& \quad \text{if } value \ast x = 0: \\
& \quad \quad \text{yield } (0, 1)
\end{align*}
\]

\[
\begin{align*}
\text{def } & \text{reduce_sum(key, vals):} \\
& \quad \text{yield } \text{sum(vals)} \\
\text{def } & \text{reduce_max(key, vals):} \\
& \quad \text{yield } \text{max(vals)}
\end{align*}
\]

- \[ \| Ax \|_1: \text{map_abs()} \rightarrow \text{reduce_sum()} \]
- \[ \| Ax \|_2^2: \text{map_square()} \rightarrow \text{reduce_sum()} \]
- \[ \| Ax \|_\infty: \text{map_abs()} \rightarrow \text{reduce_max()} \]
- \[ | Ax |_0: \text{map_zero()} \rightarrow \text{reduce_sum()} \]
Matrices and MapReduce

Ax

\| \cdot \| \|

\( A^T A \) and \( B^T A \)

QR and SVD

Conclusion
We can get a lot from $A^T A$:

- $\Sigma$: Singular values
- $V^T$: Right singular vectors
- $R$ from $QR$
We can get a lot from $A^T A$:

- $\Sigma$: Singular values
- $V^T$: Right singular vectors
- $R$ from $QR$

with a little more work...

- $U$: Left singular vectors
- $Q$ from $QR$
Computing $A^T A$ is similar to computing $||y||^2_2$.

Idea: $A^T A = \sum_{i=1}^{m} a_i^T a_i$ \hspace{1em} (a_i is the i-th row).

→ Sum of $m \times n \times n$ rank-1 matrices.

```python
1 def map(key, val):
2     # .T --> Python NumPy transpose
3     yield (0, val.T * val)
4
5 def reduce(key, vals):
6     yield (0, sum(vals))
```
$A^T A$: MapReduce
Problem: $\mathcal{O}(m)$ matrix sums on a single reducer.

Idea: have multiple reducers.
Problem: $\mathcal{O}(m)$ matrix sums on a single reducer.

Problem: need two iterations.
Need to remove communication of $O(m)$ matrices from mappers to reducers.

Idea: local partial sums on the mappers.

```python
partial_sum = zeros(n, n)
def map(key, val):
    partial_sum += val.T * val
    if key == last_key:
        yield (0, partial_sum)

def reduce(key, vals):
    yield (0, sum(vals))
```
\( A^T A: \) MapReduce

- \( O(\#(\text{mappers})) \) matrix sums on a single reducer
Suppose we are willing to have a distributed $A^T A$

Idea: emit entries of partial sums as values

```python
1  partial_sum = zeros(n, n)
2  def map(key, val):
3      partial_sum += val.T * val
4      if key == last_key:
5          for i = 1:n
6              for j = 1:n
7                  yield ((i, j), partial_sum[i, j])
8
9  def reduce(key, vals):
10     yield (key, sum(vals))
```
We want to compute $B^T A = \sum_{i=1}^{m} b_i^T a_i$

($b_i$ is i-th row of $B$, $a_i$ is i-th row of $A$)

Problem: cannot get $a_i$ and $b_i$ on the same mapper!
\[ A = \begin{bmatrix} ((1, A), [1.0, 0.0]) \\ ((2, A), [2.4, 3.7]) \\ ((3, A), [0.8, 4.2]) \\ ((4, A), [9.0, 9.0]) \end{bmatrix}, \quad B = \begin{bmatrix} ((1, B), [1.1, 3.2]) \\ ((2, B), [9.1, 0.7]) \\ ((3, B), [4.3, 2.1]) \\ ((4, B), [8.6, 2.1]) \end{bmatrix} \]

- Idea: In the map stage, use row index as key
- Problem: \( \mathcal{O}(m) \) rows communicated as data
\[ B^T A \]

\[
A = \begin{bmatrix}
((1, A), [1.0, 0.0]) \\
((2, A), [2.4, 3.7]) \\
((3, A), [0.8, 4.2]) \\
((4, A), [9.0, 9.0])
\end{bmatrix}, \quad B = \begin{bmatrix}
((1, B), [1.1, 3.2]) \\
((2, B), [9.1, 0.7]) \\
((3, B), [4.3, 2.1]) \\
((4, B), [8.6, 2.1])
\end{bmatrix}
\]

```python
def map(key, val):
    yield (key[0], (key[1], val))

def reduce(key, vals):
    # We know there are exactly two values
    (mat_id1, row1) = vals[0]
    (mat_id2, row2) = vals[1]
    if mat_id1 == 'A': yield (rand(), row2.T * row1)
    else: yield (rand(), row1.T * row2)
```
Now we have $m$ rank-1 matrices: $b_i^T a_i$, $i = 1, \ldots, m$.

Idea: Use our summation strategies from $A^T A$

```python
partial_sum = zeros(n, n)
def map(key, val):
    partial_sum += val.T * val
    if key == last_key:
        yield (0, partial_sum)
def reduce(key, vals):
    yield (0, sum(vals))
```
Problem: still $O(m)$ rows map $\rightarrow$ reduce.

Can't really get around this problem.

Result: $B^T A$ is much slower than $A^T A$. 
Matrices and MapReduce

\[ A \]

\[ A^T \]

\[ B^T \]

\[ Q \]

\[ R \]

\[ A \] and \[ B \]

\[ A^T A \] and \[ B^T A \]

QR and SVD

Conclusion
Quick QR and SVD review

Figure: $Q$, $U$, and $V$ are orthogonal matrices. $R$ is upper triangular and $\Sigma$ is diagonal with positive entries.
$A = QR$

First years: Is $R$ unique?
Tall-and-skinny (TS): $m \gg n$. $Q^T Q = I$. 
$R$ is small, so computing its SVD is cheap.
Why Tall-and-skinny QR and SVD?

1. Regression with many samples
2. Principle Component Analysis (PCA)
3. Model Reduction

Figure: Dynamic mode decomposition of a rectangular supersonic screeching jet. Joe Nichols, Stanford University.
Cholesky QR

\[ A^T A = (QR)^T (QR) = R^T Q^T QR = R^T R \]

\[ Q = AR^{-1} \]

- We already saw how to compute \( A^T A \).
- Compute \( R = \text{Cholesky}(A^T A) \) locally (cheap)
- \( AR^{-1} \) computation is similar to \( Ax \)
$AR^{-1}$
# R is available locally

def map(key, value):
    yield (key, value * inv(R))

Problem: Explicitly computing $A^T A \rightarrow$ unstable

Idea: ICME Colloquium, 4:15pm May 20, 300/300
\[ Q = AR^{-1} \]

\[ R = U_R \Sigma V^T \]

\[ A = (QU_R) \Sigma V^T = U \Sigma V^T \]

- Compute \( R = \text{Cholesky}(A^T A) \) locally (cheap)
- Compute \( R = U_R \Sigma V^T \) locally (cheap)
- \( U = A(R^{-1} U_R) \) is just an extension of \( AR^{-1} \)
$A(R^{-1}U_R)$
Matrices and MapReduce

Ax

\| \cdot \|\n
A^T A and B^T A

QR and SVD

Conclusion
Argh! These are great ideas but I do not want to implement them.

- https://github.com/arbenson/mrtsqr: Matrix computations in this talk
- Apache Mahout: machine learning library
Argh! I do not have a MapReduce cluster.

▶ icme-hadoop1.stanford.edu
Questions?

- arbenson@stanford.edu
- https://github.com/arbenson/mrtsqr