Data Structures and Performance for Scientific Computing with Hadoop and Dumbo

Austin R. Benson

Computer Sciences Division, UC-Berkeley
ICME, Stanford University

May 15, 2012

1

Dense matrix storage

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 42 & 44 \end{pmatrix}$$

How do we store the matrix in HDFS?

Dense matrix storage

$$A = \begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 42 & 44 \end{pmatrix}$$

In HDFS:

$$\langle 1, [11, 12, 13, 14] \rangle$$

$$\langle 2, [21, 22, 23, 24] \rangle$$

$$\langle 3, [31, 32, 33, 34] \rangle$$

$$\langle 4, [41, 42, 43, 44] \rangle$$

Two rows per record

or we might use:

$$\langle 1, [[11, 12, 13, 14], [21, 22, 23, 24]] \rangle$$
$$\langle 3, [[31, 32, 33, 34], [41, 42, 43, 44]] \rangle$$

Flattened list

or maybe

$$\langle 1, [11, 12, 13, 14, 21, 22, 23, 24] \rangle$$
$$\langle 3, [31, 32, 33, 34, 41, 42, 43, 44] \rangle$$

... but we do lose information here (maybe it's not important)

Full matrix

or maybe

$$\langle 1, [[11, 12, 13, 14], [21, 22, 23, 24], [31, 32, 33, 34], [41, 42, 43, 44]] \rangle$$

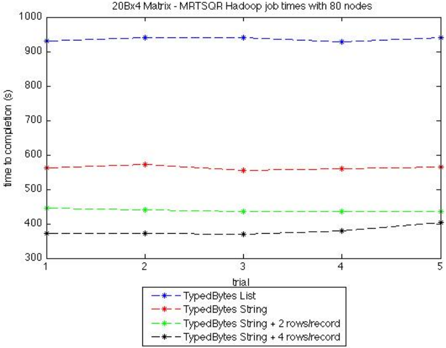
What is the "best" way?

What is the "best" way?

Depends on the application... we will look at an example later.

2

Data Serialization

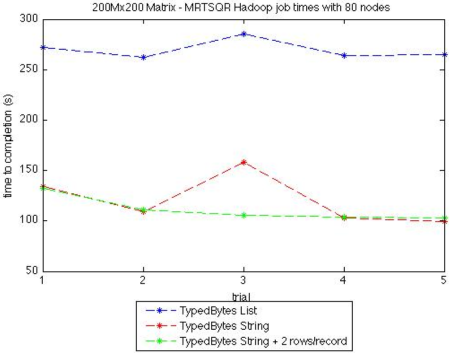


Small optimizations → 2.5x speedup!

*all data from the NERSC Magellan cluster

Data Serialization

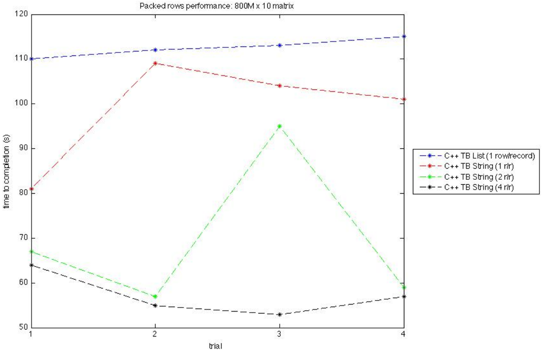
Same experiment but different matrix size (200 columns):



Again, 2.5x speedup!

Languages

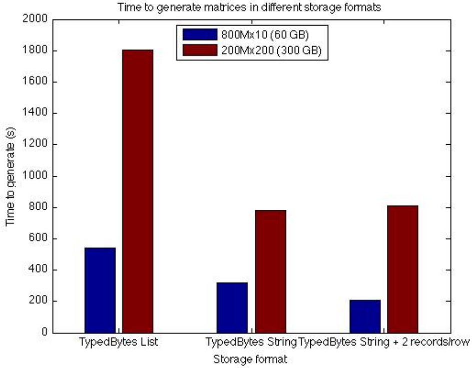
Switching from Python to C++...



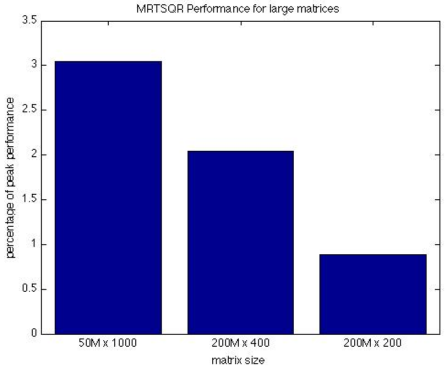
same general trend

More speedups

Algorithm performance isn't the only place where we see speedups



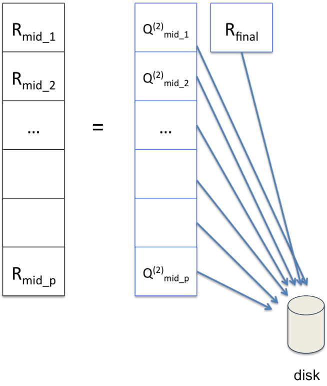
Why can we expect these speedups?



These are *not* high-performance implementations. We care about I/O performance.

3

Suppose we need to write many small matrices to disk.



Code

Code:

```
git clone git://github.com/icme/mapreduce-workshop.git
cd mapreduce-workshop/arbenson
```

Files:

- speed_test.py (tester)
- small_matrix_test.py (driver)

```
# Naive
for matrix in matrices_to_store:
    for row in matrix:
        yield key, row

# Store <key, matrix> pairs.
for matrix in matrices_to_store:
    yield key, matrix

# Store the matrix as a flat data structure
for matrix in matrices_to_store:
    yield key, [entry for row in matrix for entry in row]

# Use python's struct module
for matrix in matrices_to_store:
    flat = [entry for row in matrix for entry in row]
    yield key, struct.pack('d'*len(flat), *flat)
```

Writing a key-value pair for each row

```
# Naive
for matrix in matrices_to_store:
    for row in matrix:
        yield key, row

# Store <key, matrix> pairs.
for matrix in matrices_to_store:
    yield key, matrix

# Store the matrix as a flat data structure
for matrix in matrices_to_store:
    yield key, [entry for row in matrix for entry in row]

# Use python's struct module
for matrix in matrices_to_store:
    flat = [entry for row in matrix for entry in row]
    yield key, struct.pack('d'*len(flat), *flat)
```

If we do not need row-specific keys, store one key per matrix

```
# Naive
for matrix in matrices_to_store:
    for row in matrix:
        yield key, row

# Store <key, matrix> pairs.
for matrix in matrices_to_store:
    yield key, matrix

# Store the matrix as a flat data structure
for matrix in matrices_to_store:
    yield key, [entry for row in matrix for entry in row]

# Use python's struct module
for matrix in matrices_to_store:
    flat = [entry for row in matrix for entry in row]
    yield key, struct.pack('d'*len(flat), *flat)
```

Store as a flat list

```python
# Naive
for matrix in matrices_to_store:
    for row in matrix:
        yield key, row

# Store <key, matrix> pairs.
for matrix in matrices_to_store:
    yield key, matrix

# Store the matrix as a flat data structure
for matrix in matrices_to_store:
    yield key, [entry for row in matrix for entry in row]

# Use python's struct module
for matrix in matrices_to_store:
    flat = [entry for row in matrix for entry in row]
    yield key, struct.pack('d'*len(flat), *flat)
```

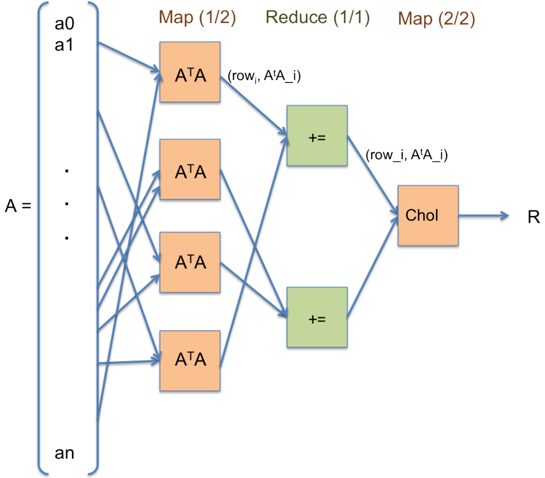
Pack as straight bytes

4

Algorithm

Cholesky QR: R $=$ chol($A^T A$, 'upper')

Implementation for MapReduce



Mapper implementation

Which of these implementations is better?

```
# mapper
A = []
for key, row in input:
   A += row
for i, row in enum(AᵀA):
    yield i, row
```

```
# mapper
A = []
def compress():
  for i, row in enum(AᵀA):
    yield i, row
  A = []

block = 100
for key, row in input:
   A += row
   if len(A > block):
      compress()
compress()
```

Mapper implementation

Which of these implementations is better?

```
# mapper
A = []
for key, row in input:
   A += row
for i, row in enum(AᵀA):
    yield i, row
```

```
# mapper
A = []
def compress():
  for i, row in enum(AᵀA):
    yield i, row
  A = []

block = 100
for key, row in input:
   A += row
   if len(A > block):
      compress()
compress()
```

Answer: the one on the left (usually)

Why?

1. Shuffle time
2. Reduce bottleneck

However, the left implementation could run out of memory.

Mapper implementation

Can we do better? Yes

```
# mapper
A_loc = zeros(ncols, ncols)
A = []
def compress():
  A_loc += AᵀA
  A = []

block = 100
for key, row in input:
   A += row
   if len(A > block):
      compress()
compress()
for i, row in enumerate
(A_loc):
    yield i, row
```

Questions?

Austin R. Benson
arbenson@gmail.com
https://github.com/arbenson/mrtsqr