Alchemist: An Apache Spark to MPI Interface

Alex Gittens, Kai Rothauge, Michael W. Mahoney, Shusen Wang, Michael Ringenburg, Kristyn Maschhoff, Prabhat, Lisa Gerhardt, Jey Kottalam

CUG2018
Stockholm
Why should MPI codes interface with Spark?

Google trends popularity: MPI vs Hadoop
Q: What about less embarrassingly parallel computations?
A: Use Spark and MPI
Example: linear algebra in Spark

**Pros for Spark:**

- Faster development, easier reuse
- One abstract uniform interface (RDD)
- An entire ecosystem that can be used before and after the NLA computations
- Spark can take advantage of available local linear algebra codes
- Automatic fault-tolerance, out-of-core support

**Pros for MPI:** Classical MPI-based linear algebra implementations will be faster and more efficient
**Motivation**

- **NERSC**: Spark for data-centric workloads and scientific analytics
- **RISELab**: characterization of linear algebra in Spark (MLlib, MLMatrix)
- **Cray**: customers demand for Spark; understand performance concerns
Data parallel programming model
- Resilient distributed datasets (RDDs); optionally cached in memory
- Driver forms DAG, schedules tasks on executors
Spark Communication

Bulk Synchronous Programming Model:
- Each overall job (DAG) broken into stages
- Stages broken into parallel, independent tasks
- Communication happens only between stages
Spark Overheads: the view of one task

\[
\text{task start delay} = \text{(time between stage start and when driver sends task to executor)}
\]

\[
\text{scheduler delay} = \text{(time between task being sent and time starts deserializing) + (time between task result serialization and driver receiving task's completion message)}
\]

\[
\text{task overhead time} = \text{(fetch wait time) + (executor deserialize time) + (result serialization time) + (shuffle write time)}
\]

\[
\text{time waiting until stage end} = \text{(time waiting for final task in stage to end)}
\]
Cori Phase I—NERSC supercomputer—specs:

- 1630 compute nodes,
- 128 GB/node, 32 2.3GHz Haswell cores/node
- Lustre Filesystem, Aries interconnect

### Running times for NMF and PCA

<table>
<thead>
<tr>
<th>Nodes / cores</th>
<th>MPI Time</th>
<th>Spark Time</th>
<th>Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NMF</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 / 1,600</td>
<td>1 min 6 s</td>
<td>4 min 38 s</td>
<td>4.2x</td>
</tr>
<tr>
<td>100 / 3,200</td>
<td>45 s</td>
<td>3 min 27 s</td>
<td>4.6x</td>
</tr>
<tr>
<td>300 / 9,600</td>
<td>30 s</td>
<td>70 s</td>
<td>2.3x</td>
</tr>
<tr>
<td><strong>PCA (2.2TB)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 / 3,200</td>
<td>1 min 34 s</td>
<td>15 min 34 s</td>
<td>9.9x</td>
</tr>
<tr>
<td>300 / 9,600</td>
<td>1 min</td>
<td>13 min 47 s</td>
<td>13.8x</td>
</tr>
<tr>
<td>500 / 16,000</td>
<td>56 s</td>
<td>19 min 20 s</td>
<td>20.7x</td>
</tr>
<tr>
<td><strong>PCA (16TB)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI: 1,600 / 51,200</td>
<td>2 min 40 s</td>
<td>69 min 35 s</td>
<td>26x</td>
</tr>
<tr>
<td>Spark: 1,522 / 48,704</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
With favorable data (tall and skinny) and well-adapted algorithms, **Spark LA is 2x-26x slower than MPI when IO is included**

**Spark overheads are orders of magnitude higher than the computations** in PCA (a typical iterative algorithm)

The large gaps in performance suggests **interfacing MPI-based codes with Spark**
The Next Step: **Alchemist**

- Since Spark is 4+x slower than MPI, propose sending the matrices to MPI codes, then receiving the results.
- For efficiency, want as little overhead as possible (File I/O, RAM, network usage, computational efficiency).

Alternative approaches:

1. Write to HDFS: *slow file I/O, manual data layout*
2. Other MPI-Spark bridges: *assume sparse data sets, use RAM disk, or write to file*
3. Apache Ignite (and Alluxio, etc.): *requires using C/C++ interfaces, manual data layout, extra copy in memory, TCP/IP*

**Alchemist:**

Uses *in-memory transfer*, transparently *provides data relayout*, explicitly *handles dense data sets*.
Main Challenges

- **Minimizing communication time** between Spark workers and Alchemist workers

- **Switching between the matrix distribution schemes** imposed by Spark and MPI codes, as needed

Our approach:
1. Communicate using row-partitioned matrices
2. Relayout only on the Alchemist side
3. Use Elemental library to handle implicit and explicit relayout
Launch Alchemist before Spark

```
srun -N $(ALCHEMISTNODECOUNT) -n $$((ALCHEMISTNODECOUNT*32/OMP_NUM_THREADS)) -c $OMP_NUM_THREADS -w $SPARK_WORKER_DIR/hosts.alchemist. --output=$SPARK_WORKER_DIR/alchemistIOs/stdout_%t.log --error=$SPARK_WORKER_DIR/alchemistIOs/stderr_%t.log ./core/target/alchemist &
```

Start a Spark job
Start an Alchemist context in your Spark job

```
val conf = new SparkConf().setAppName("Alchemist ADMM KRR Test")
val sc = new SparkContext(conf)
val al = new Alchemist(sc)
```
Alchemist: A User’s View

Create IndexedRowMatrix RDDs
Send over to Alchemist and store handles

```scala
val alMatA = AlMatrix(al, Ardd)
val alMatB = AlMatrix(al, Brdd)
```

Manipulate using Alchemist MPI interface and store handles

```scala
val alMatX = al.SkilarkADMMKRR(alMatA, alMatB, regression, lossfunction, regularizer, kernel, kernelparam, kernelparam2, kernelparam3, lambda, maxiter, tolerance, rho, seed, randomfeatures, numfeaturepartitions)
```

Retrieve results to IndexedRowMatrix RDDs

```scala
var solXrdd = alMatX.getIndexedRowMatrix()
```
Communication Times (1)

- Random **Tall-and-Skinny 400GB** matrix (5.12M-by-10K)
- Spark to Alchemist communication time (s)

<table>
<thead>
<tr>
<th>Spark Worker Nodes</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>40</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>62.1</td>
<td>65.2</td>
<td>66.4</td>
<td>72.4</td>
<td>72.8</td>
<td>76.7</td>
<td>88.5</td>
</tr>
<tr>
<td>16</td>
<td>75.6</td>
<td><strong>68.3</strong></td>
<td>72.8</td>
<td>81.1</td>
<td>89.3</td>
<td>93.5</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>73</td>
<td>69.7</td>
<td><strong>62.8</strong></td>
<td>77.5</td>
<td>82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>78.5</td>
<td>75.4</td>
<td>69.8</td>
<td><strong>66.8</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>69.6</td>
<td>65.4</td>
<td>62.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>70.6</td>
<td>67.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>64.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Random **Short-and-Fat 400GB** matrix (5.12M-by-10K)
- Spark to Alchemist communication time (s)

<table>
<thead>
<tr>
<th>Spark Worker Nodes</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>32</th>
<th>40</th>
<th>48</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>59.8</td>
<td>50.0</td>
<td>38.5</td>
<td>30.1</td>
<td>15.2</td>
<td>14.7</td>
<td>18.6</td>
</tr>
<tr>
<td>16</td>
<td>55.8</td>
<td>34.0</td>
<td>24.5</td>
<td>20.2</td>
<td>13.9</td>
<td>13.8</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>56.2</td>
<td>34.9</td>
<td>21.9</td>
<td>18.0</td>
<td></td>
<td>12.4</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>54.1</td>
<td>30.5</td>
<td>22.1</td>
<td>15.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>52.9</td>
<td>30.6</td>
<td></td>
<td>22.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>54.5</td>
<td></td>
<td>27.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>57.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operations Implemented</td>
<td>Library/Memory Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Send</td>
<td>- / 1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Retrieve</td>
<td>- / 1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Transpose</td>
<td>Elemental / 2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>Elemental / 2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KMeans</td>
<td>- / 1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVD</td>
<td>Elemental / 2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Truncated SVD</td>
<td>ARPACK / 2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSQR linear solver</td>
<td>LibSkylark / 1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regularized CG linear solver</td>
<td>LibSkylark / 1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel Solver (reg/clas, regularized)</td>
<td>LibSkylark / 1X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDF5 Reader</td>
<td>- / 2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Application: Matrix Multiplication

Impractical in Spark:
- Matrices/RDDs are row-partitioned
- One must be converted to column-partitioned
- **Requires an all-to-all shuffle that often fails** once the matrix is distributed

```scala
// Spark Matrix Multiply
val sparkMatC = sparkMatA.toBlockMatrix()
  .multiply(sparkMatB.toBlockMatrix()).toIndexedRowMatrix

// Alchemist Matrix Multiply
val alMatA = AlMatrix(al, sparkMatA)
val alMatB = AlMatrix(al, sparkMatB)
val alMatC = al.matMul(alMatA, alMatB)
val alRes = alMatC.getIndexedRowMatrix()
```
### Application: Matrix Multiplication

<table>
<thead>
<tr>
<th>GB/nodes</th>
<th>Send</th>
<th>Mult</th>
<th>Receive</th>
<th>Alchemist</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8/1</td>
<td>5.90±2.17</td>
<td>6.60±0.07</td>
<td>2.19±0.58</td>
<td>14.68±2.69s</td>
<td>160.31±8.89s</td>
</tr>
<tr>
<td>12/1</td>
<td>16.66±0.88</td>
<td>75.69±0.42</td>
<td>19.43±0.45</td>
<td>111.78±1.26s</td>
<td>809.31±51.9s</td>
</tr>
<tr>
<td>56/2</td>
<td>32.50±2.88</td>
<td>178.68±24.8</td>
<td>55.83±0.37</td>
<td>267.02±27.38s</td>
<td>-Failed-</td>
</tr>
<tr>
<td>144/4</td>
<td>69.40±1.85</td>
<td>171.73±0.81</td>
<td>66.80±3.46</td>
<td>307.94±4.57s</td>
<td>-Failed-</td>
</tr>
</tbody>
</table>

- Generated random matrices and used same number of Spark and Alchemist nodes
- Take-away: **Spark’s multiply is slow even on one executor, and unreliable once there are more**
Application: SVD

Compare **Alchemist** wrapper around **ARPACK** with **MLlib**

Compute rank-20 decomposition of random matrices

**22 Spark nodes** vs **8 Alchemist nodes** (16 workers/node)

```scala
val alMatA = AlMatrix(al, rdd)
val (alU, alS, alV) = al.truncatedSVD(alMatA, k)
val alUreturned = alU.getIndexedRowMatrix()
val alSreturned = alS.getIndexedRowMatrix()
val alVreturned = alV.getIndexedRowMatrix()
```
Compare **Alchemist** wrapper around **ARPACK** with **MLlib**

**Compute rank-20 decomposition** of random matrices

**22 Spark nodes** vs **8 Alchemist nodes** (16 workers/node)
Compare **Alchemist** wrapper around **ARPACK** with **MLlib**

**Compute rank-20 decomposition** of random matrices

**22 Spark nodes vs 8 Alchemist nodes** (16 workers/node)
Future Work

- PySpark interface
- Container support
- GEMM for row-partitioned matrix (avoid 2x memory overhead)
- ScaLAPack redistribution support
Thank you

https://github.com/alexgittens/alchemist