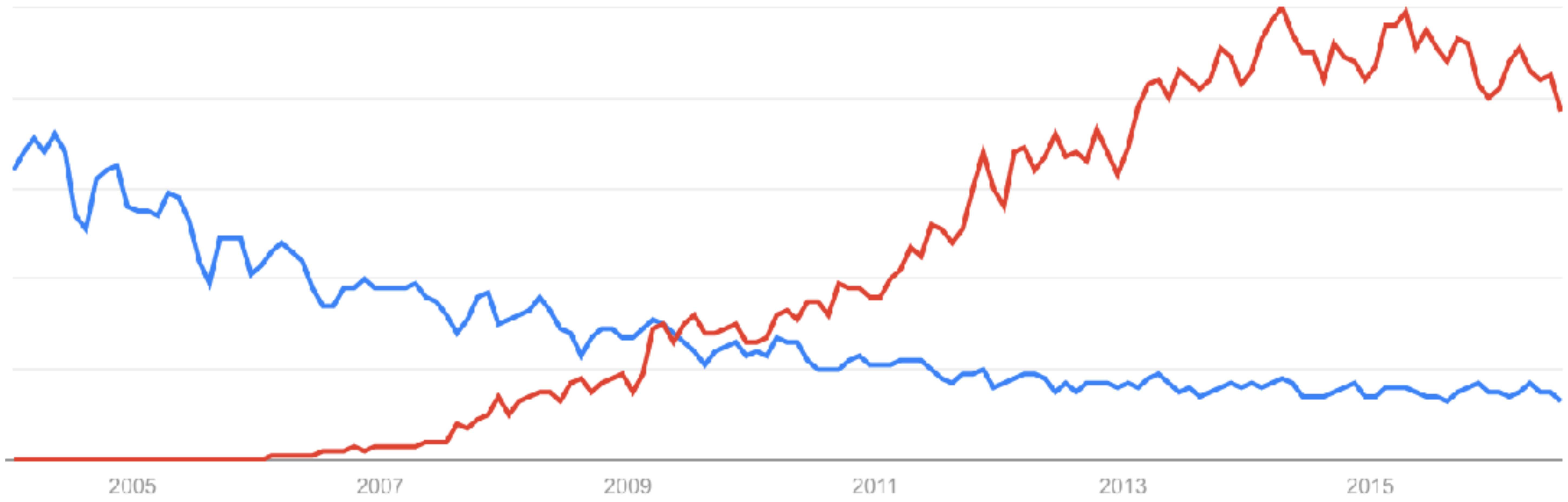
Alchemist: An Apache Spark to MPI Interface

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

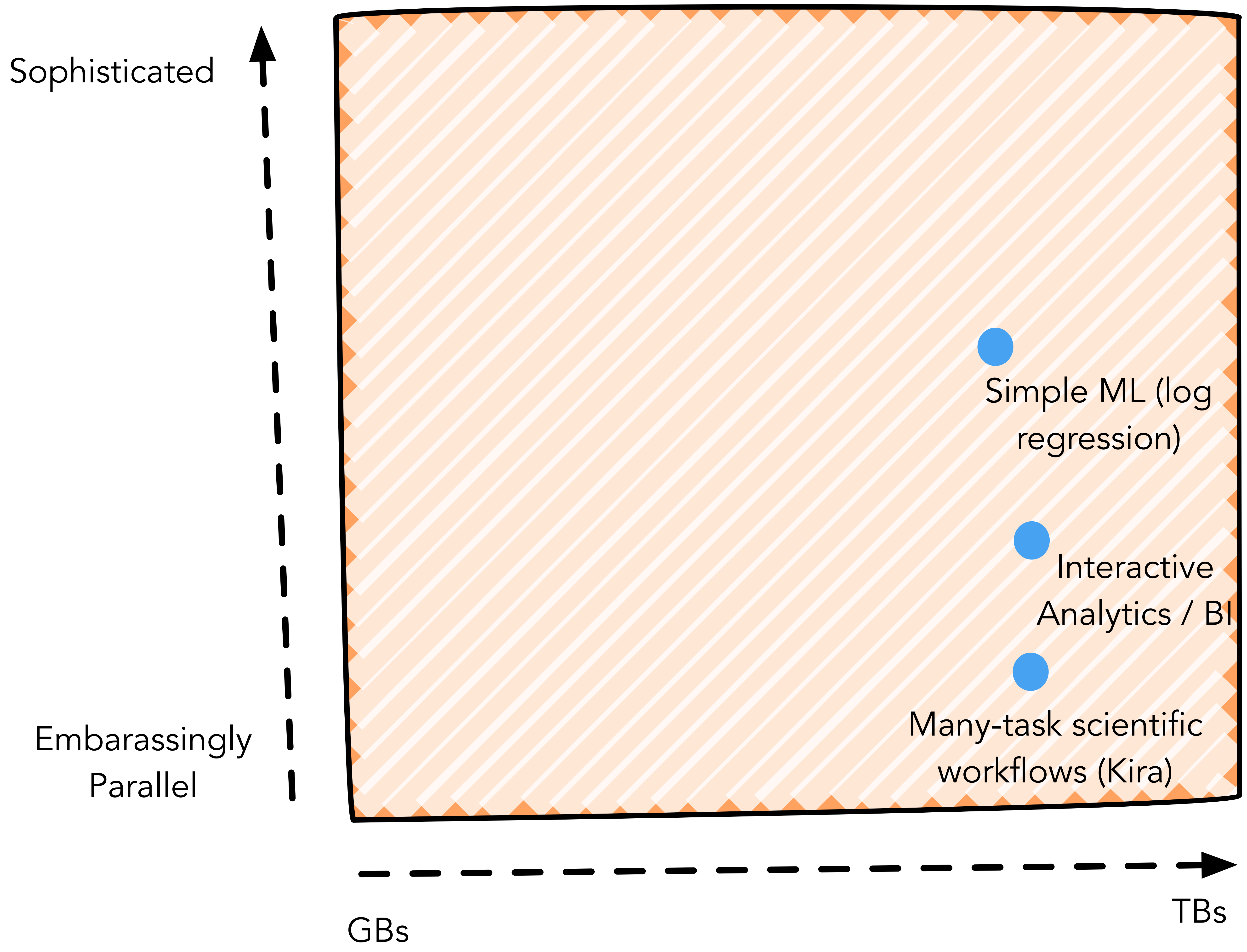
CUG2018
Stockholm

Why should MPI codes interface with Spark?



Google trends popularity: MPI vs Hadoop

Spark Use Cases



Q: What about less embarrassingly parallel computations?

A: Use Spark and MPI

Example: linear algebra in Spark

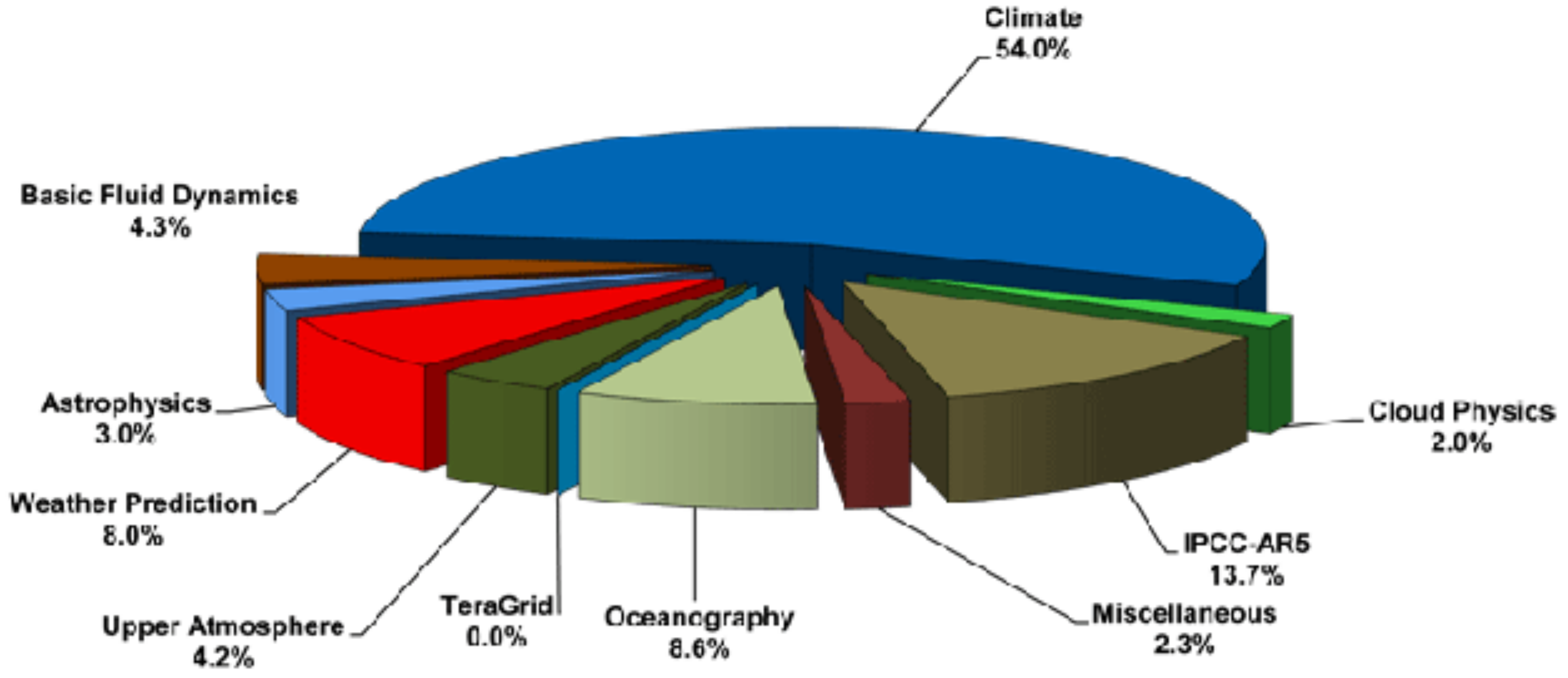
Pros for Spark:

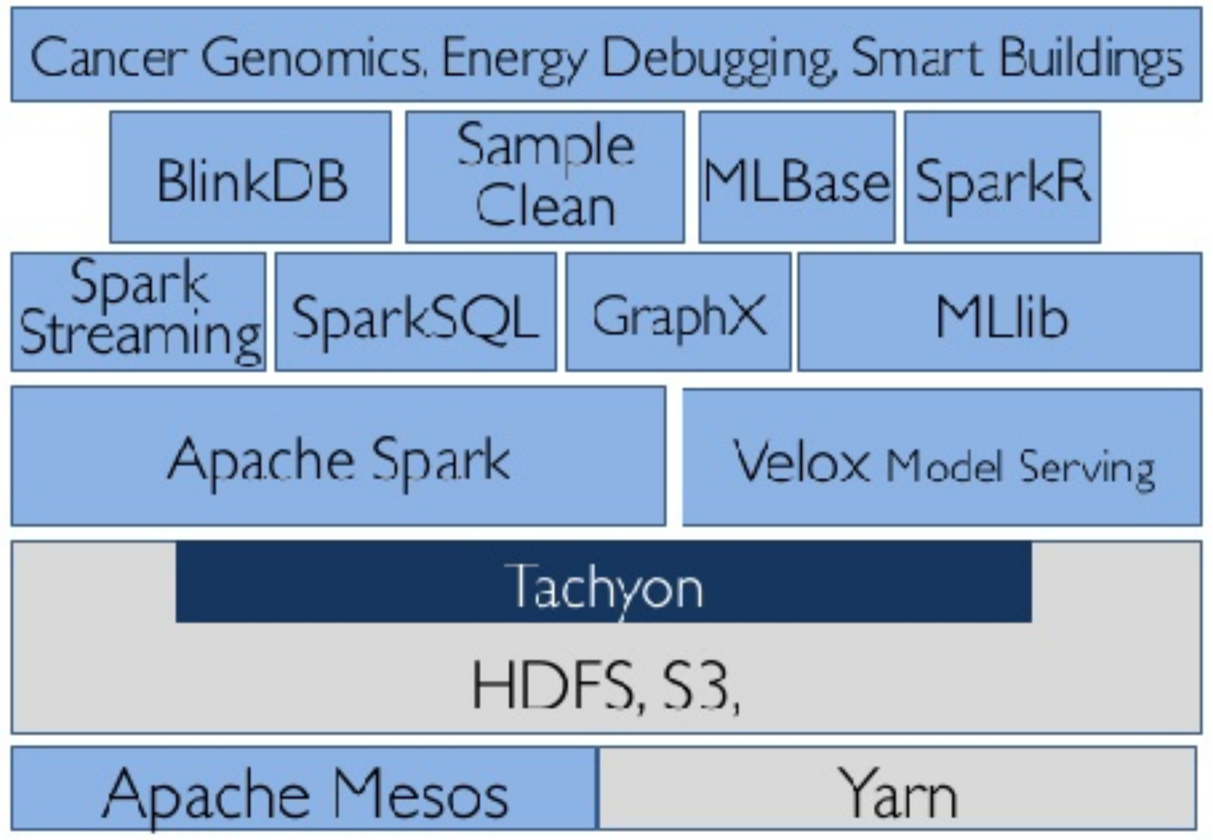
- Faster development, easier reuse
- One abstract uniform interface (RDD)
- An entire ecosystem that can be used before and after the NLA computations
- Spark can take advantage of available local linear algebra codes
- Automatic fault-tolerance, out-of-core support

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





Cancer Genomics, Energy Debugging, Smart Buildings

BlinkDB | Sample Clean | MLBase | SparkR

Spark Streaming | SparkSQL | GraphX | MLlib

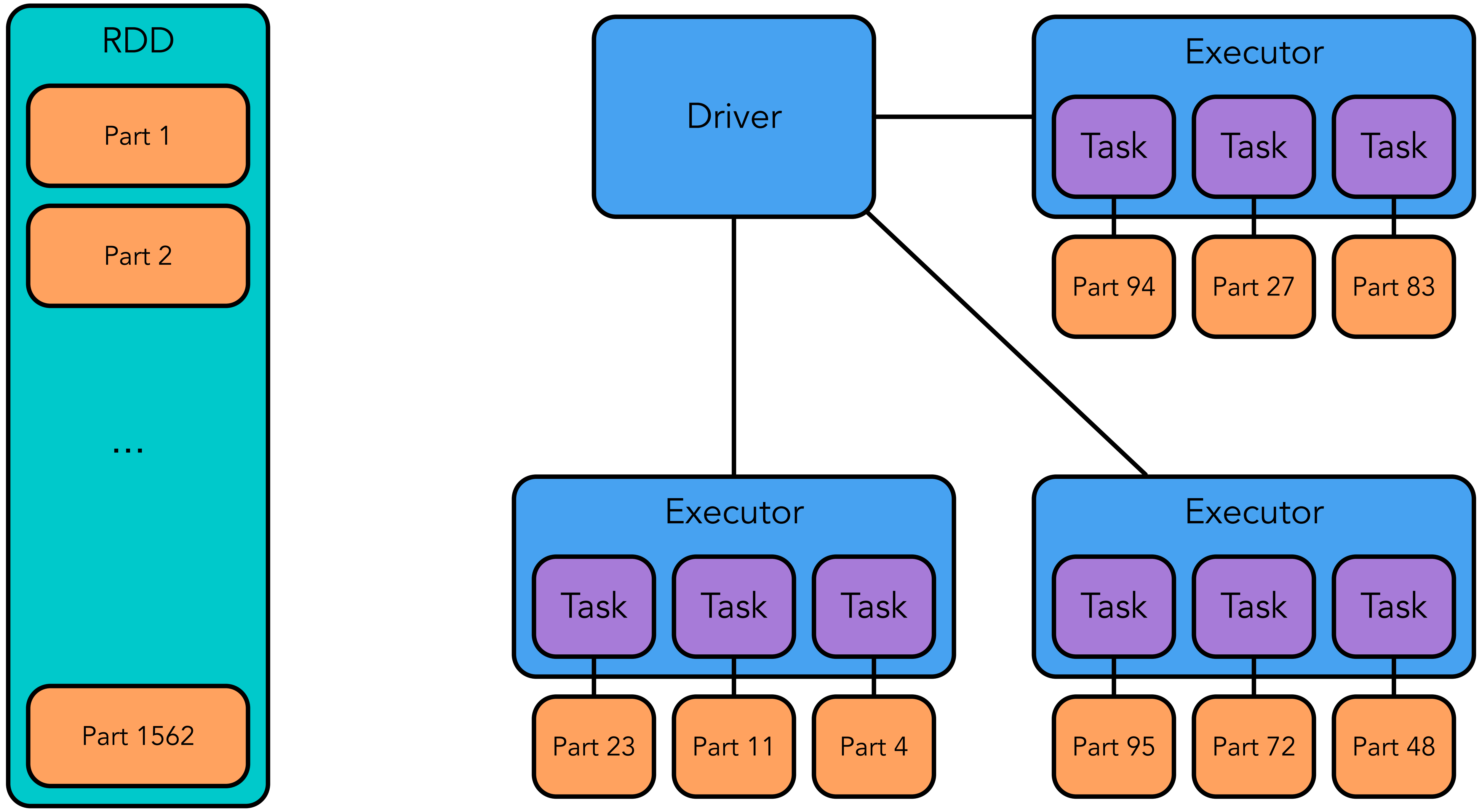
Apache Spark | Velox Model Serving

Tachyon

HDFS, S3,

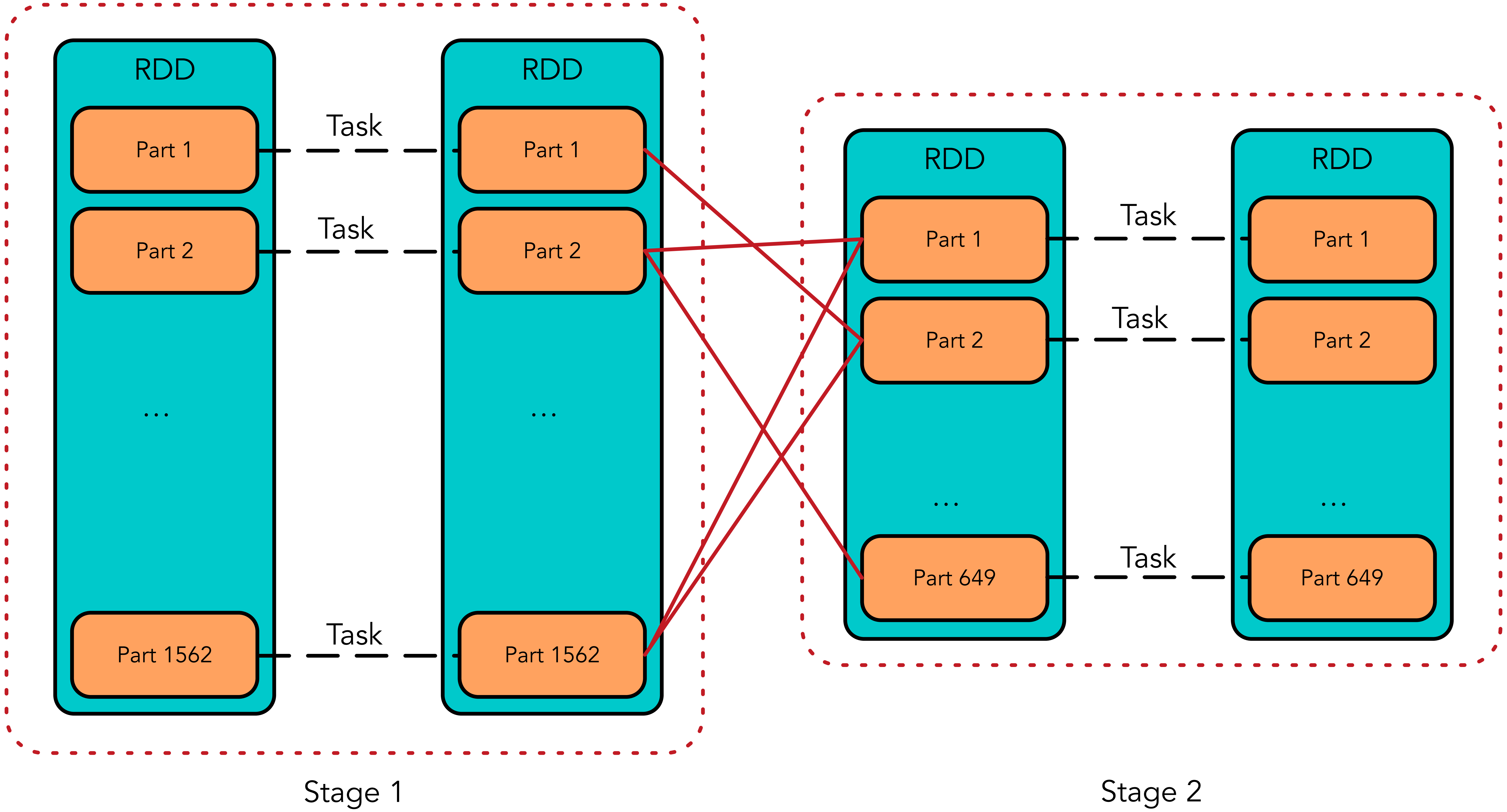
Apache Mesos | Yarn

Spark Architecture



- Data parallel programming model
- Resilient distributed datasets (RDDs); optionally cached in memory
- Driver forms DAG, schedules tasks on executors

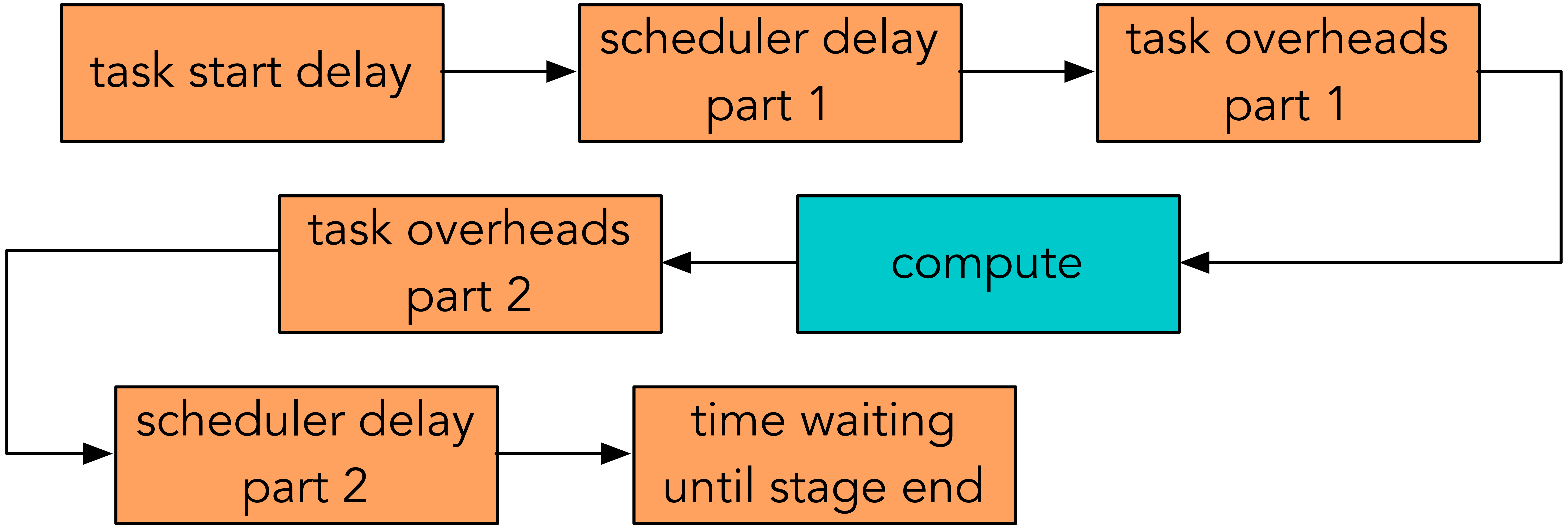
Spark Communication



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



task start delay

scheduler delay part 1

task overheads part 1

compute

task overheads part 2

scheduler delay part 2

time waiting until stage end

task start delay = (time between stage start and when driver sends task to executor)

scheduler delay = (time between task being sent and time starts deserializing)+ (time between task result serialization and driver receiving task's completion message)

task overhead time = (fetch wait time) + (executor deserialize time) + (result serialization time) + (shuffle write time)

time waiting until stage end = (time waiting for final task in stage to end)

Running times for NMF and PCA

Cori Phase I—NERSC supercomputer—specs:

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

	Nodes / cores	**MPI Time**	**Spark Time**	**Gap**
NMF	50 / 1,600	1 min 6 s	4 min 38 s	4.2x
	100 / 3,200	45 s	3 min 27 s	4.6x
	300 / 9,600	30 s	70 s	2.3x
PCA (2.2TB)	100 / 3,200	1 min 34 s	15 min 34 s	9.9x
	300 / 9,600	1 min	13 min 47 s	13.8x
	500 / 16,000	56 s	19 min 20 s	20.7x
PCA (16TB)	MPI: 1,600 / 51,200 Spark: 1,522 / 48,704	2 min 40 s	69 min 35 s	26x

MPI vs Spark: Lessons Learned

- 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)

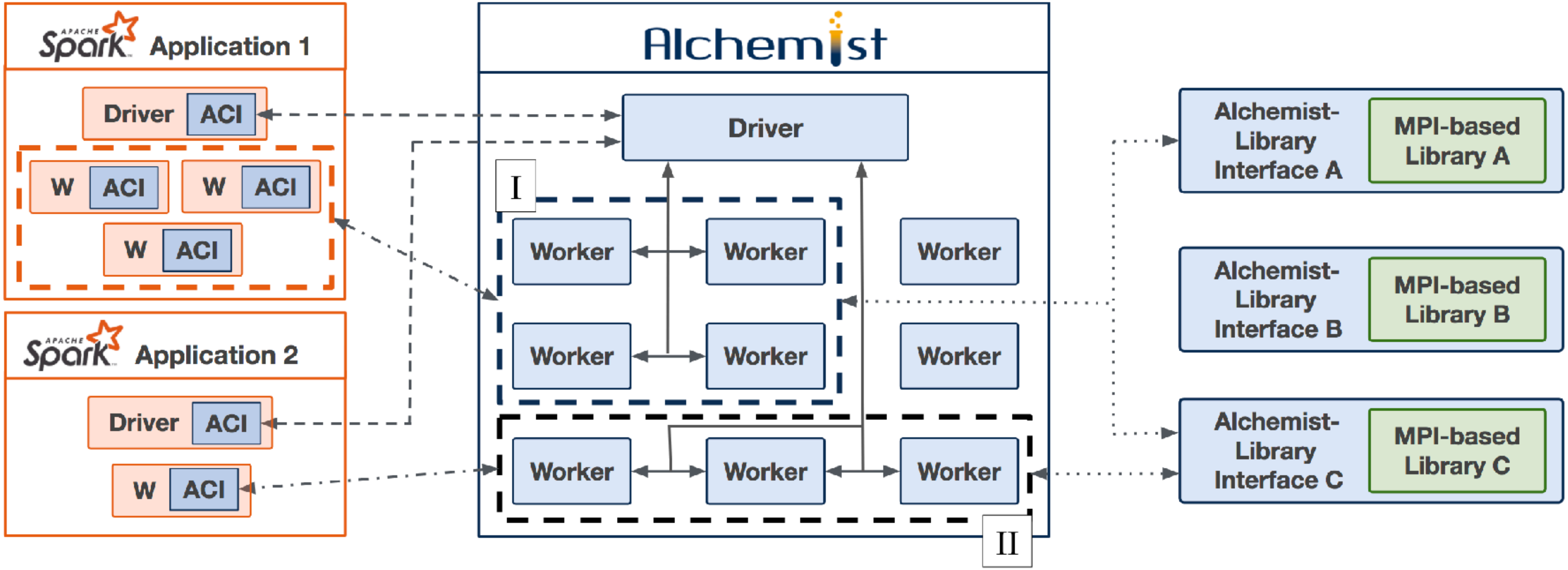
Alternative approaches:

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***

Alchemist Architecture



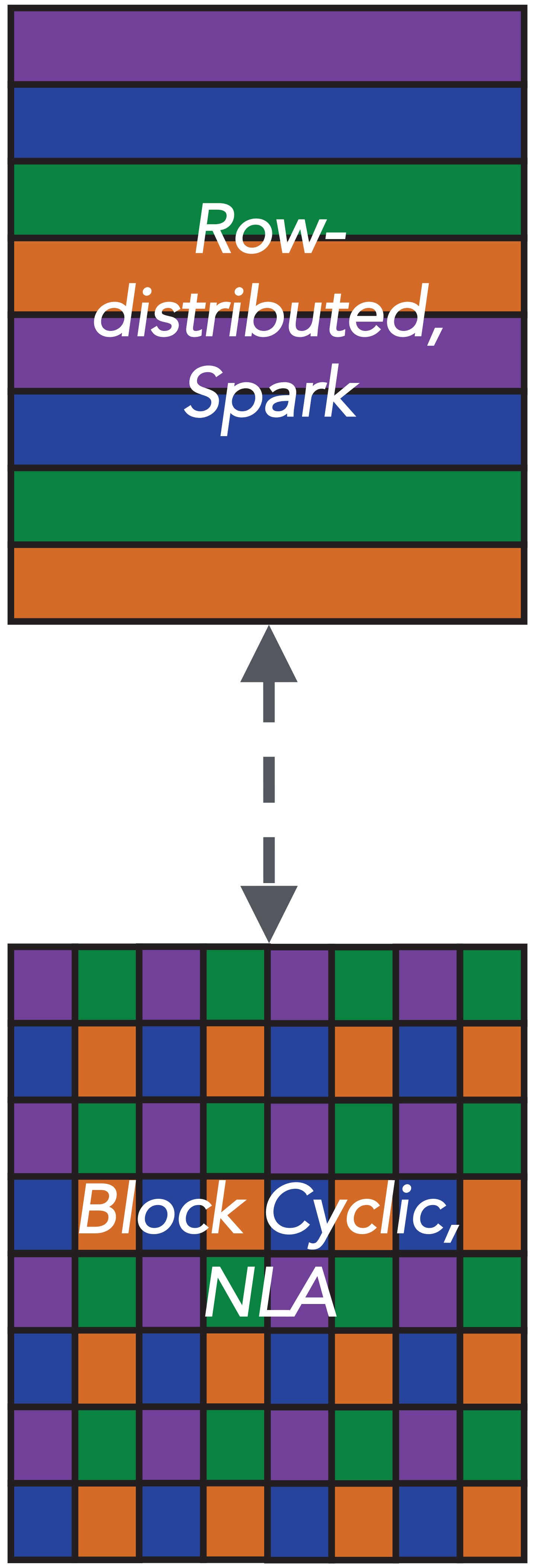
◀- - - -▶ Inter-Driver Socket Communication	◀· · · · ·▶ Dynamic linking
◀- · - · -▶ Inter-Worker Socket Communication	◀———▶ MPI Communication

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

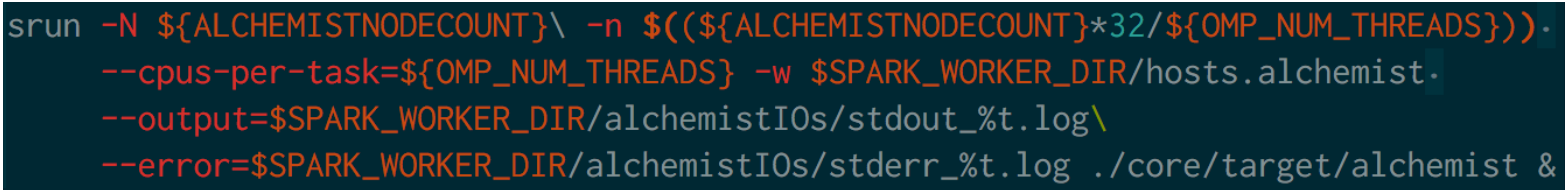
Our approach:

1. Communicate using row-partitioned matrices
2. Relayout only on the Alchemist side
3. Use `Elemental` library to handle implicit and explicit relayout



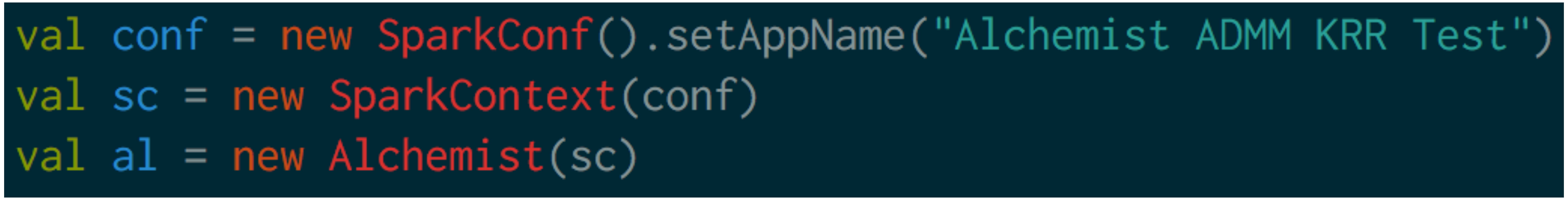
Alchemist: A User’s View

Launch Alchemist before Spark



```
srun -N ${ALCHEMISTNODECOUNT}\ -n $((${ALCHEMISTNODECOUNT}*32/${OMP_NUM_THREADS}))·
     --cpus-per-task=${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 &
```

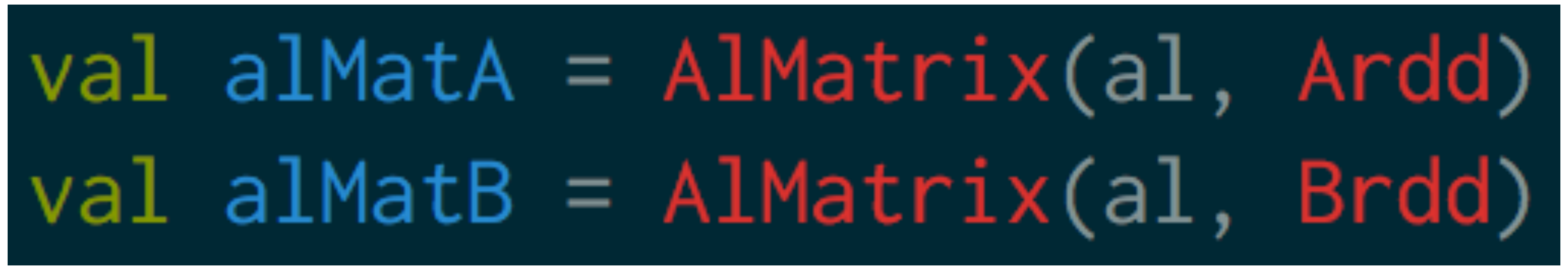
Start a Spark job
Start an Alchemist context in your Spark job



```
val conf = new SparkConf().setAppName("Alchemist ADMM KRR Test")
val sc = new SparkContext(conf)
val al = new Alchemist(sc)
```

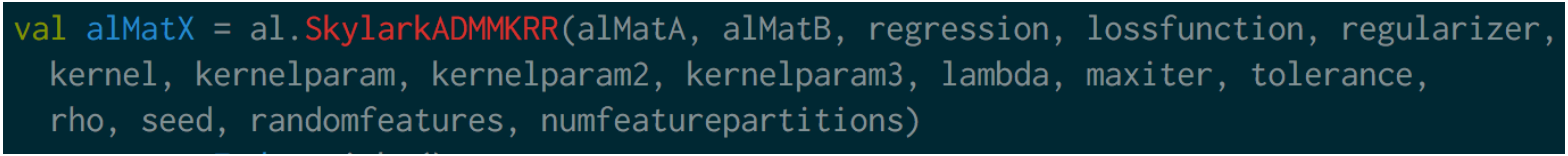
Alchemist: A User’s View

Create IndexedRowMatrix RDDs
Send over to Alchemist and store handles



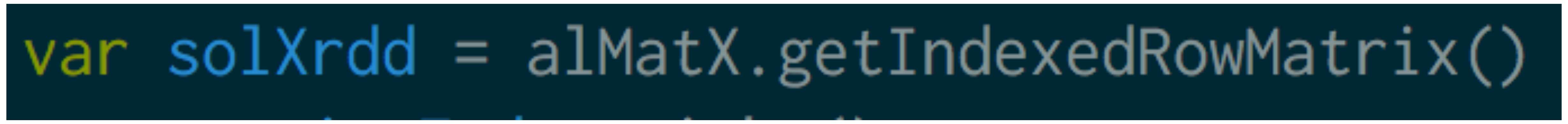
```
val alMatA = AlMatrix(al, Ardd)
val alMatB = AlMatrix(al, Brdd)
```

Manipulate using Alchemist MPI interface and store handles



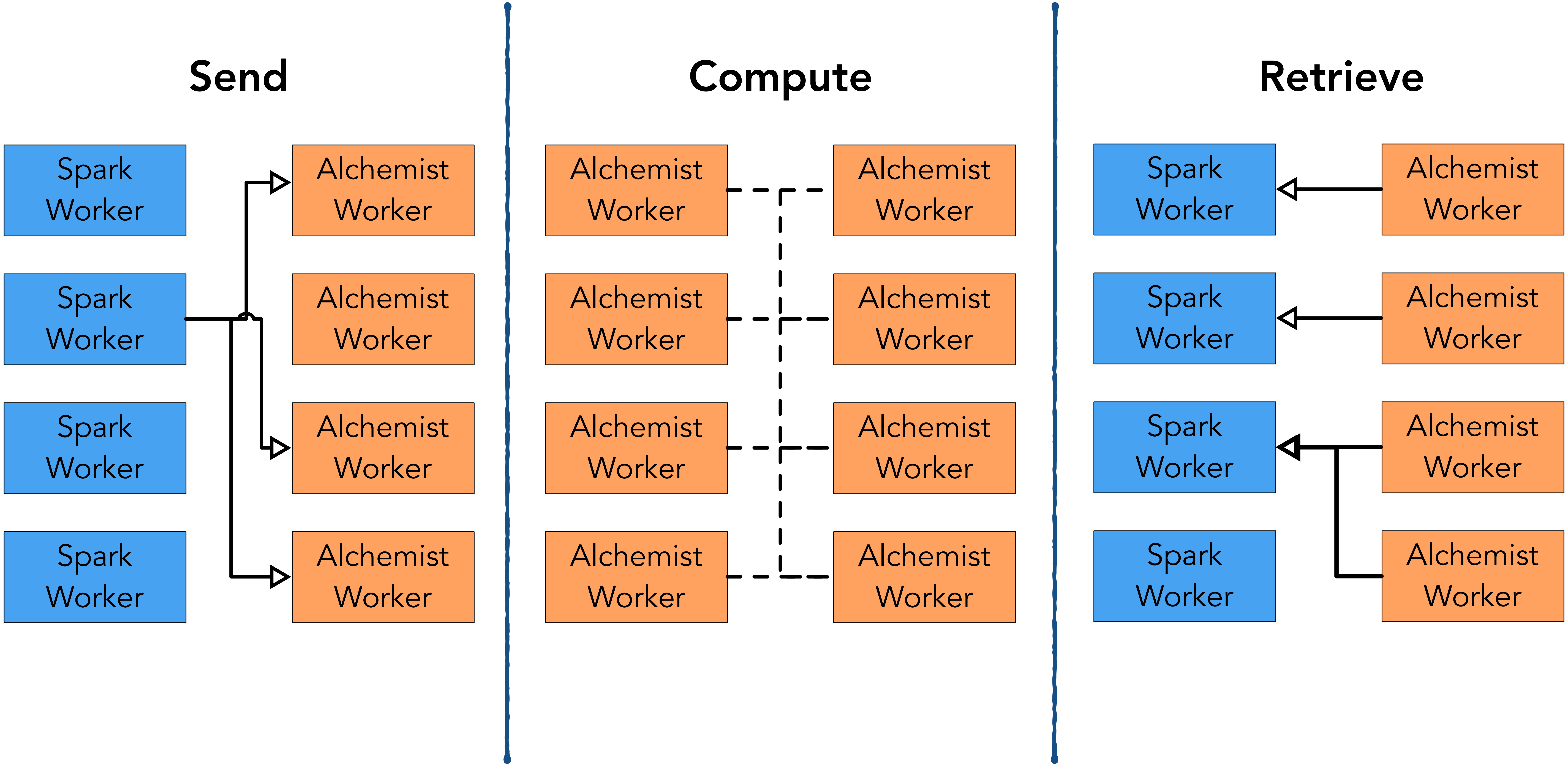
```
val alMatX = al.SkylarkADMMKRR(alMatA, alMatB, regression, lossfunction, regularizer,
  kernel, kernelparam, kernelparam2, kernelparam3, lambda, maxiter, tolerance,
  rho, seed, randomfeatures, numfeaturepartitions)
```

Retrieve results to IndexedRowMatrix RDDs



```
var solXrdd = alMatX.getIndexedRowMatrix()
```

Communication Scheme



Send

Spark Worker
Spark Worker
Spark Worker
Spark Worker

Alchemist Worker
Alchemist Worker
Alchemist Worker
Alchemist Worker

Compute

Alchemist Worker
Alchemist Worker
Alchemist Worker
Alchemist Worker

Alchemist Worker
Alchemist Worker
Alchemist Worker
Alchemist Worker

Retrieve

Spark Worker
Spark Worker
Spark Worker
Spark Worker

Alchemist Worker
Alchemist Worker
Alchemist Worker
Alchemist Worker

Communication Times (1)

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

Alchemist Worker Nodes

Spark Worker Nodes

	8	**16**	**24**	**32**	**40**	**48**	**56**
8	**62.1**	65.2	66.4	72.4	72.8	76.7	88.5
16	75.6	**68.3**	72.8	81.1	89.3	93.5	
24	73	69.7	**62.8**	77.5	82		
32	78.5	75.4	69.8	**66.8**			
40	69.6	65.4	62.4				
48	70.6	67.9					
56	64.5						

Communication Times (2)

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

Alchemist Worker Nodes

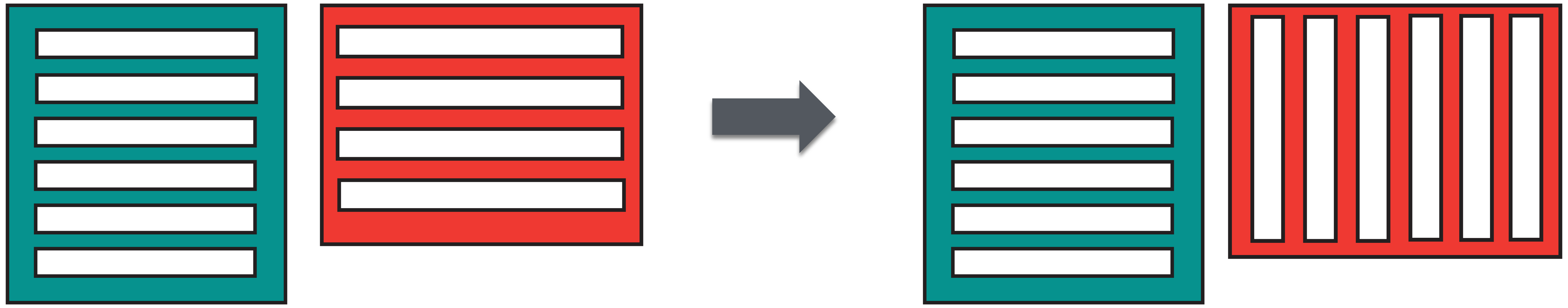
Spark Worker Nodes

	8	**16**	**24**	**32**	**40**	**48**	**56**
8	59.8	50.0	38.5	30.1	15.2	**14.7**	18.6
16	55.8	34.0	24.5	20.2	13.9	**13.8**	
24	56.2	34.9	21.9	18.0	**12.4**		
32	54.1	30.5	22.1	**15.1**			
40	52.9	30.6	**22.7**				
48	54.5	**27.2**					
56	**57.6**						

Currently Interfaced Codes

Operations Implemented	**Library/Memory Cost**
Matrix **Send**	- / 1X
Matrix **Retrieve**	- / 1X
Matrix **Transpose**	Elemental / 2X
Matrix **Multiply**	Elemental / 2X
KMeans	- / 1X
SVD	Elemental / 2X
Truncated SVD	ARPACK / 2X
LSQR **linear solver**	LibSkylark / 1X
Regularized **CG linear solver**	LibSkylark / 1X
Kernel Solver (reg/clas, regularized)	LibSkylark / 1X
HDF5 Reader	- / 2X

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

```
// 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

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

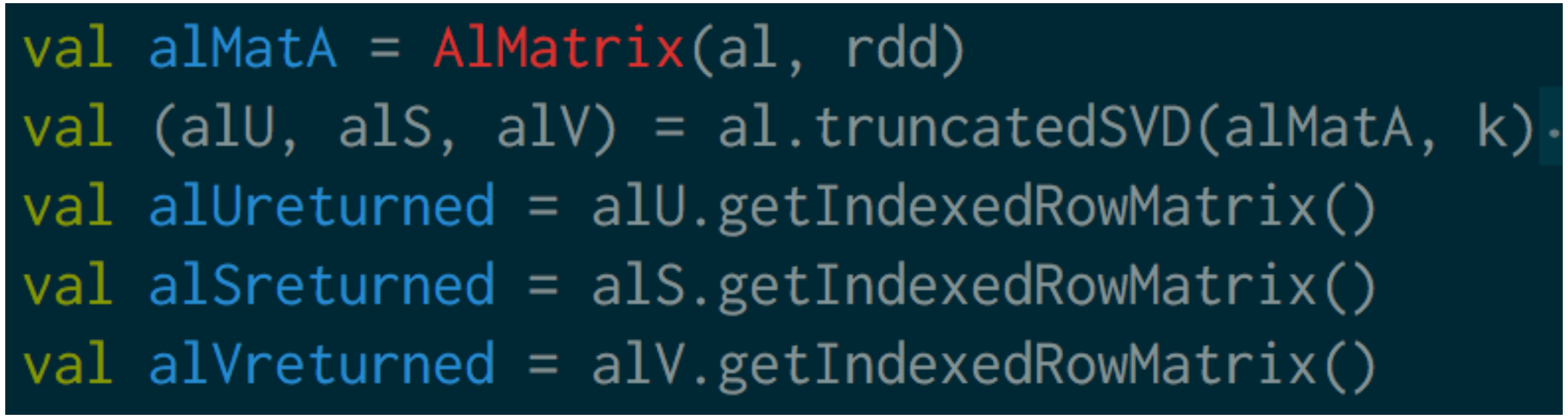
- Generated random matrices and used same number of Spark and Alchemist nodes
- Take-away: ***Spark's multiply is slow even on one executor, and unreliable once there are more***

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)



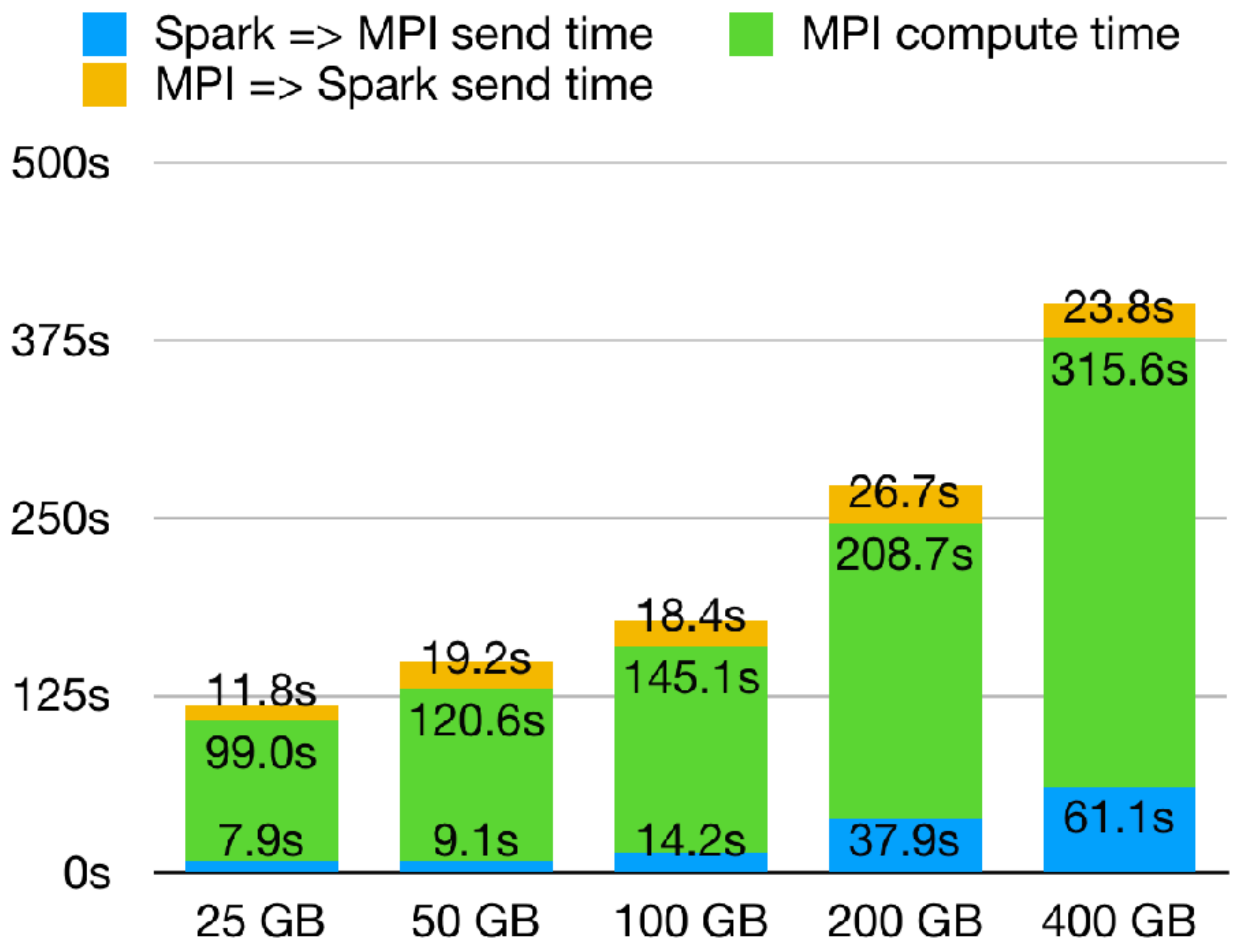
```
val alMatA = AlMatrix(al, rdd)
val (alU, alS, alV) = al.truncatedSVD(alMatA, k)
val alUreturned = alU.getIndexedRowMatrix()
val alSreturned = alS.getIndexedRowMatrix()
val alVreturned = alV.getIndexedRowMatrix()
```

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)

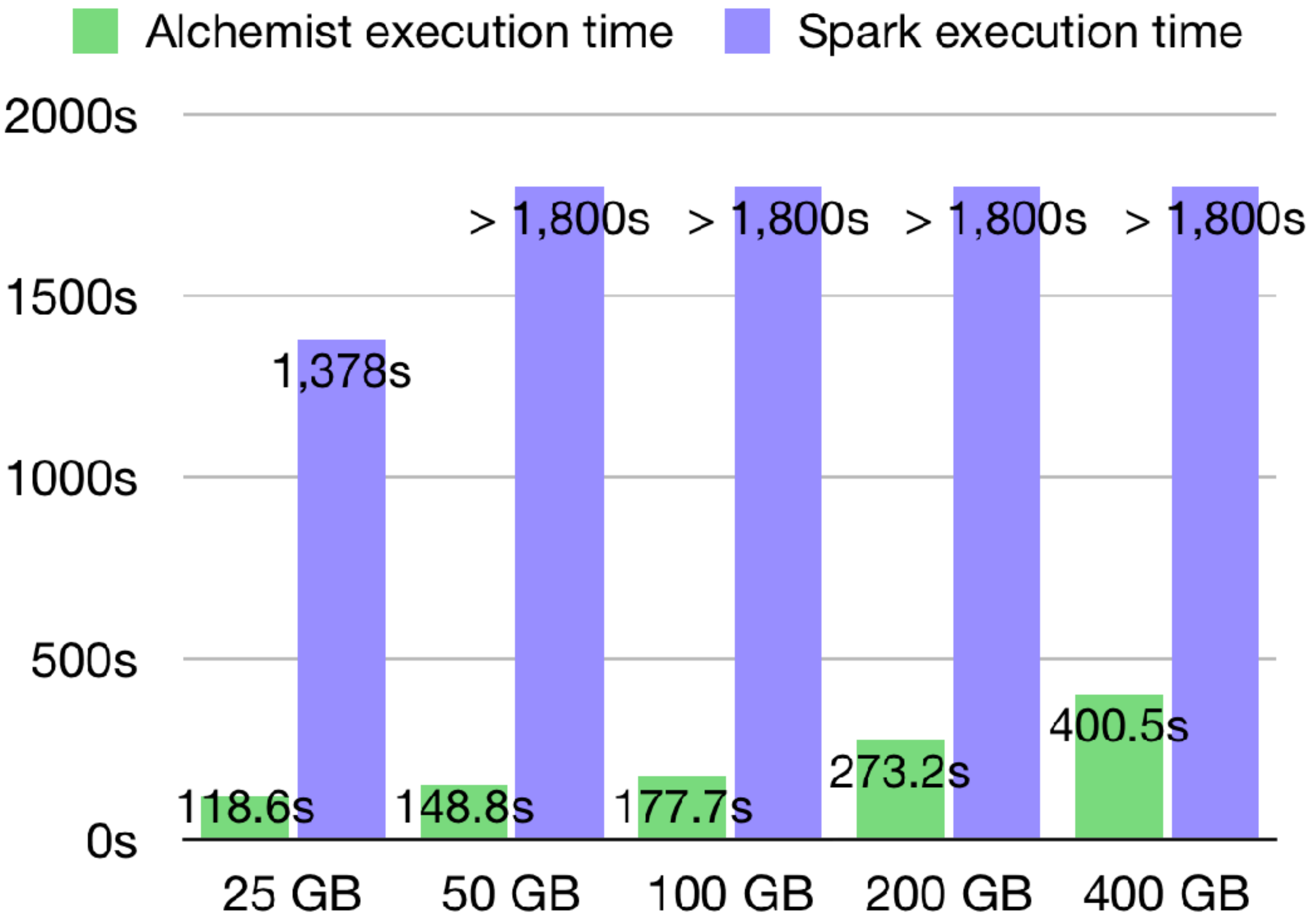


Application: SVD

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

Compute rank-20 decomposition of random matrices

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



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