Running Alchemist on Cray XC and CS Series Supercomputers:

Dask and PySpark Interfaces, Deployment Options, and Data Transfer Times

Kristyn Maschhoff, Ph.D., Cray Inc.

CRAY™

riselab UC Berkeley

Running Alchemist on Cray XC and CS Series Supercomputers:

Dask and PySpark Interfaces, Deployment Options, and Data Transfer Times

Kai Rothauge, Haripriya Ayyalasomayajula, Kristyn J. Maschhoff, Michael Ringenburg, Michael W. Mahoney

Talk Overview

- Introduction
- Python, Dask and PySpark Interfaces
- Alchemist Containers
- Summary
- Q&A

- Not covered today, but included in paper
  - Discussion of communication overheads

- Apache Spark shown to have significant overheads when performing linear algebra computations compared to MPI-based implementations
- **Alchemist** interfaces between Apache Spark and high-performance computing (HPC) libraries
- Idea:
  - Use Spark for regular data analysis workflow
  - When computationally intensive calculations are required, call relevant MPI-based codes from Spark using Alchemist, send results to Spark
- Combines ***high productivity*** of Spark with ***high performance*** of MPI

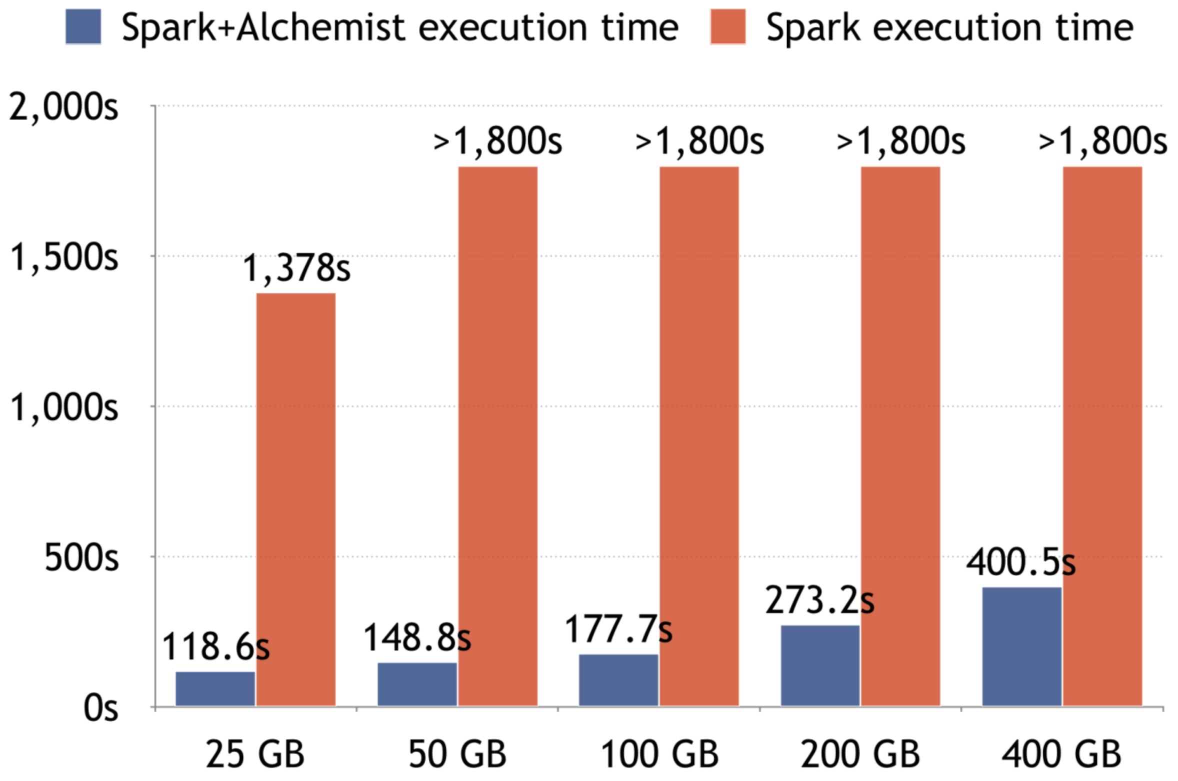
Example: Truncated SVD

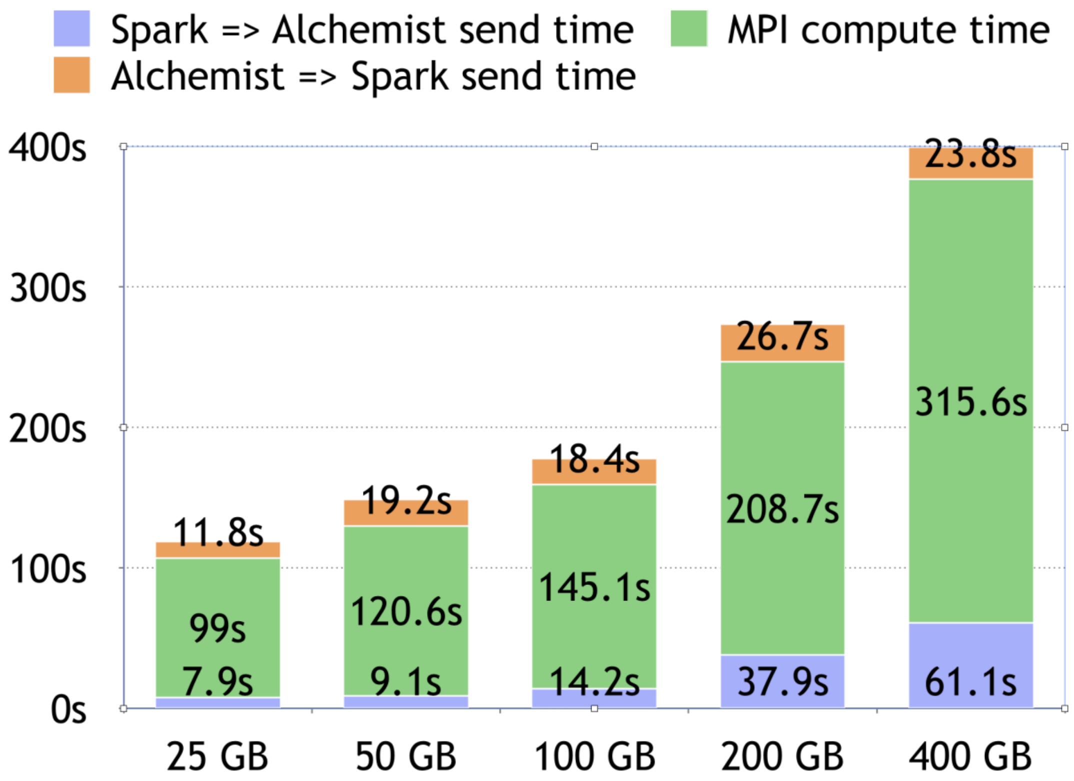
- Use Alchemist and Spark's MLlib to get rank-20 truncated SVD
- Experiments run on NERSC supercomputer Cori (a Cray XC40) - each node of Cori has 128GB RAM and 32 cores.

Example: Truncated SVD

Experiment Setup:

- Spark: 22 nodes; Alchemist: 8 nodes
- A: m-by-10K, where m = 5M, 2.5M, 1.25M, 625K, 312.5K
- Ran jobs for at most 30 minutes (1800 s)





Target users:

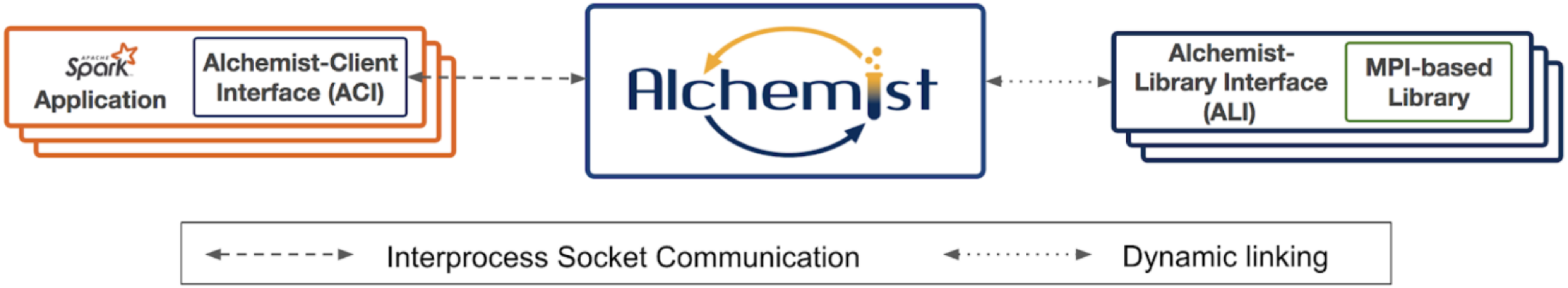
- ***Scientific & engineering communities:***

  Use Spark for analysis of large scientific datasets or computationally intensive workflows by calling existing or custom HPC libraries where appropriate

- ***Machine learning practitioners*** and ***data analysts***:

  Better performance of a wide range of large-scale, computationally intensive ML and data analysis algorithms (principal component analysis, recommender systems, leverage scores, etc.)

Basic Framework



Spark Application
Alchemist-Client Interface (ACI)
Alchemist
Alchemist-Library Interface (ALI)
MPI-based Library
Interprocess Socket Communication
Dynamic linking

- **Alchemist**: Acts as bridge between the Spark application and HPC libraries
- **Alchemist-Client Interface (ACI):** API for user; communicates with Alchemist via TCP/IP sockets; responsible for data serialization and deserialization
- **Alchemist-Library Interface (ALI):** Shared object, imports HPC library, provides generic interface for Alchemist to communicate with HPC libraries

New Features

- Server-based architecture
- Support for sparse data sets
- Docker image

  Github: project-alchemist/AlchemistDocker

  Docker Hub: projectalchemist/alchemist

- New Alchemist-Client Interfaces (ACI):

Github:	project-alchemist/ACIPython
Github:	project-alchemist/ACIDask
Github:	project-alchemist/ACIPySpark

Python, Dask and PySpark Interfaces

python™ Interface

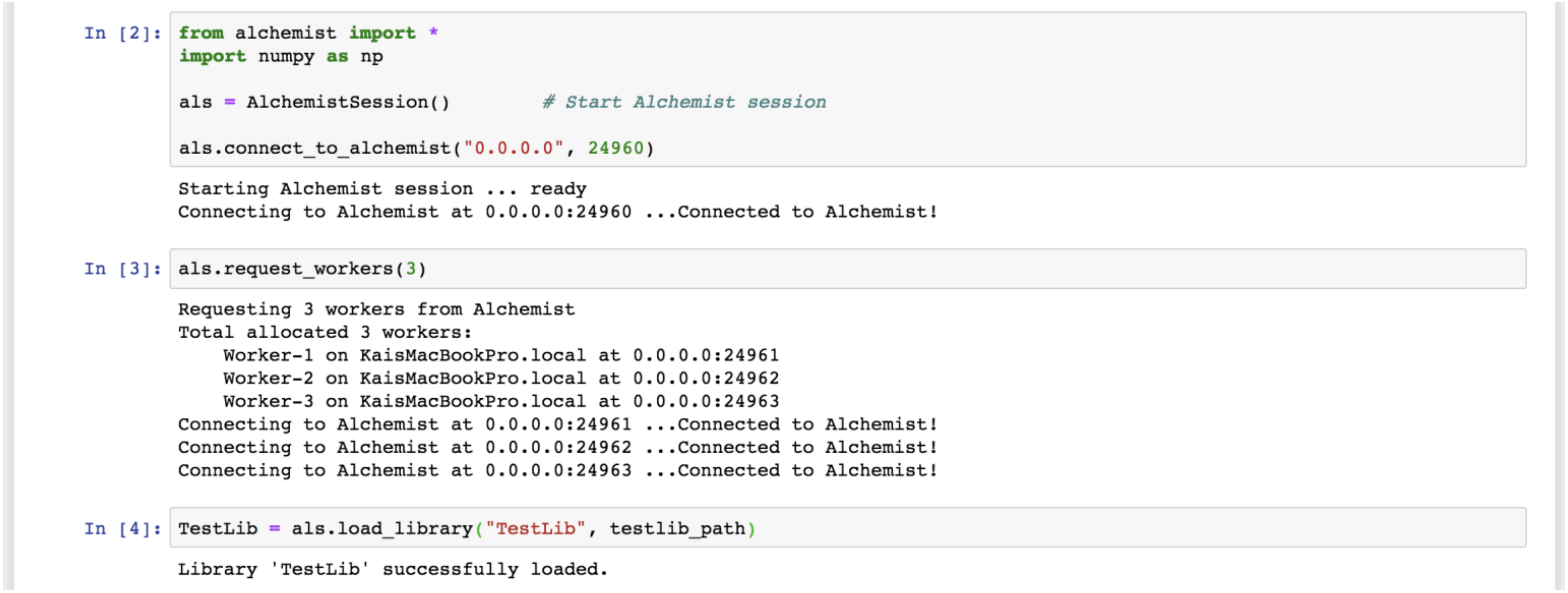
Python is the most popular language for data analysis and machine learning tasks

ACIPython allows Python users to connect to Alchemist and make use of existing HPC libraries for their data analysis and machine learning needs

Design of ACIPython resembles that of the Spark interface:

- User imports ACIPython and uses it to connect to Alchemist and request a certain number of workers
- Communication is primarily with the Alchemist driver, but large matrices (or other large data sets) are sent directly to the Alchemist workers

Interface



```
In [2]: from alchemist import *
        import numpy as np

        als = AlchemistSession()          # Start Alchemist session

        als.connect_to_alchemist("0.0.0.0", 24960)
```

```
Starting Alchemist session ... ready
Connecting to Alchemist at 0.0.0.0:24960 ...Connected to Alchemist!
```

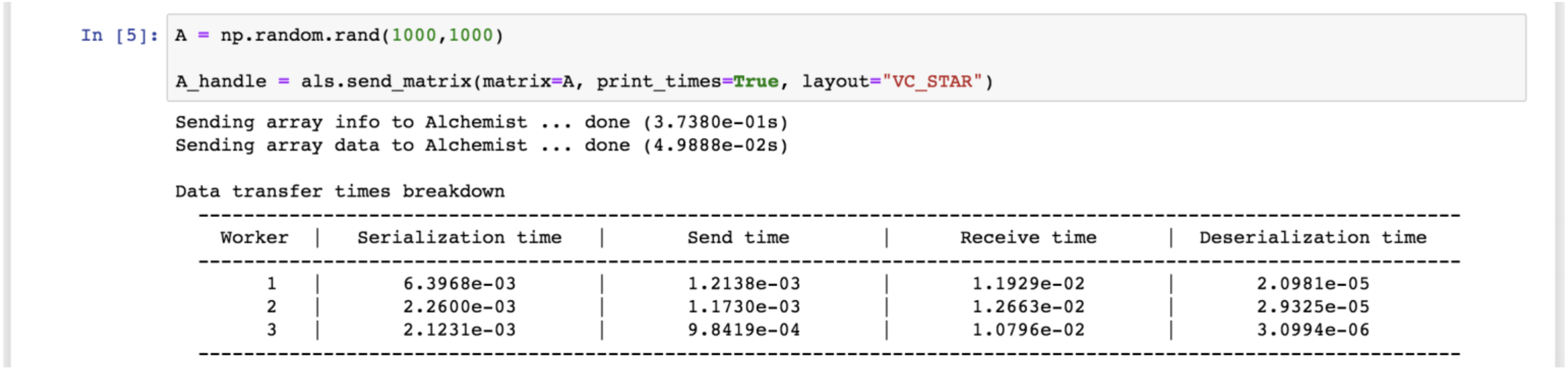
```
In [3]: als.request_workers(3)
```

```
Requesting 3 workers from Alchemist
Total allocated 3 workers:
    Worker-1 on KaisMacBookPro.local at 0.0.0.0:24961
    Worker-2 on KaisMacBookPro.local at 0.0.0.0:24962
    Worker-3 on KaisMacBookPro.local at 0.0.0.0:24963
Connecting to Alchemist at 0.0.0.0:24961 ...Connected to Alchemist!
Connecting to Alchemist at 0.0.0.0:24962 ...Connected to Alchemist!
Connecting to Alchemist at 0.0.0.0:24963 ...Connected to Alchemist!
```

```
In [4]: TestLib = als.load_library("TestLib", testlib_path)
```

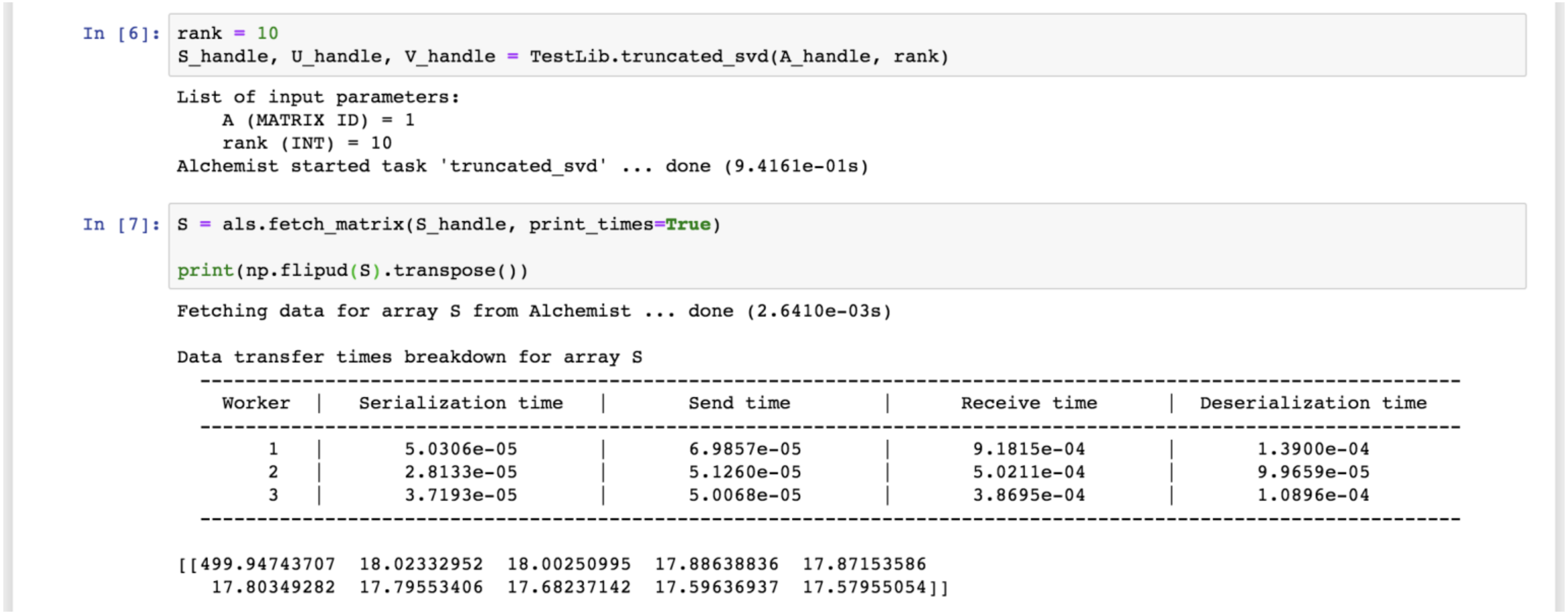
```
Library 'TestLib' successfully loaded.
```

python™ Interface



```
In [5]: A = np.random.rand(1000,1000)

        A_handle = als.send_matrix(matrix=A, print_times=True, layout="VC_STAR")

Sending array info to Alchemist ... done (3.7380e-01s)
Sending array data to Alchemist ... done (4.9888e-02s)

Data transfer times breakdown
  ----------------------------------------------------------------------------------------------------------
   Worker |   Serialization time    |        Send time        |      Receive time       |  Deserialization time
  ----------------------------------------------------------------------------------------------------------
      1   |       6.3968e-03        |       1.2138e-03        |       1.1929e-02        |       2.0981e-05
      2   |       2.2600e-03        |       1.1730e-03        |       1.2663e-02        |       2.9325e-05
      3   |       2.1231e-03        |       9.8419e-04        |       1.0796e-02        |       3.0994e-06
  ----------------------------------------------------------------------------------------------------------
```

python™ Interface



```
In [6]: rank = 10
        S_handle, U_handle, V_handle = TestLib.truncated_svd(A_handle, rank)

        List of input parameters:
            A (MATRIX ID) = 1
            rank (INT) = 10
        Alchemist started task 'truncated_svd' ... done (9.4161e-01s)

In [7]: S = als.fetch_matrix(S_handle, print_times=True)

        print(np.flipud(S).transpose())

        Fetching data for array S from Alchemist ... done (2.6410e-03s)

        Data transfer times breakdown for array S
          ----------------------------------------------------------------------------------------------------------------
           Worker  |   Serialization time   |         Send time         |        Receive time        |  Deserialization time
          ----------------------------------------------------------------------------------------------------------------
              1    |      5.0306e-05        |        6.9857e-05         |        9.1815e-04          |      1.3900e-04
              2    |      2.8133e-05        |        5.1260e-05         |        5.0211e-04          |      9.9659e-05
              3    |      3.7193e-05        |        5.0068e-05         |        3.8695e-04          |      1.0896e-04
          ----------------------------------------------------------------------------------------------------------------

        [[499.94743707  18.02332952  18.00250995  17.88638836  17.87153586
           17.80349282  17.79553406  17.68237142  17.59636937  17.57955054]]
```

Interface

- ACIPython interface assumes that the underlying application is running on a single process, i.e. ACIPython does not support distributed data sets
- Use cases:
  - Data can be loaded from file by Alchemist directly
  - The client application can load or generate data that is too large to fit in memory in chunks, and then transfer each chunk to Alchemist
  - Some computations generate large data sets during intermediate stages of computation that have to be stored as distributed matrices, but the input and output data sets may easily fit inside the memory of a single node
- Use Dask or PySpark interfaces for distributed data sets

DASK Interface

Dask is a popular scalable data analytics platform for Python

- Data structures such as arrays and dataframes for storing data in larger-than-memory or distributed environments
- Dynamic task schedulers that are optimized for computation

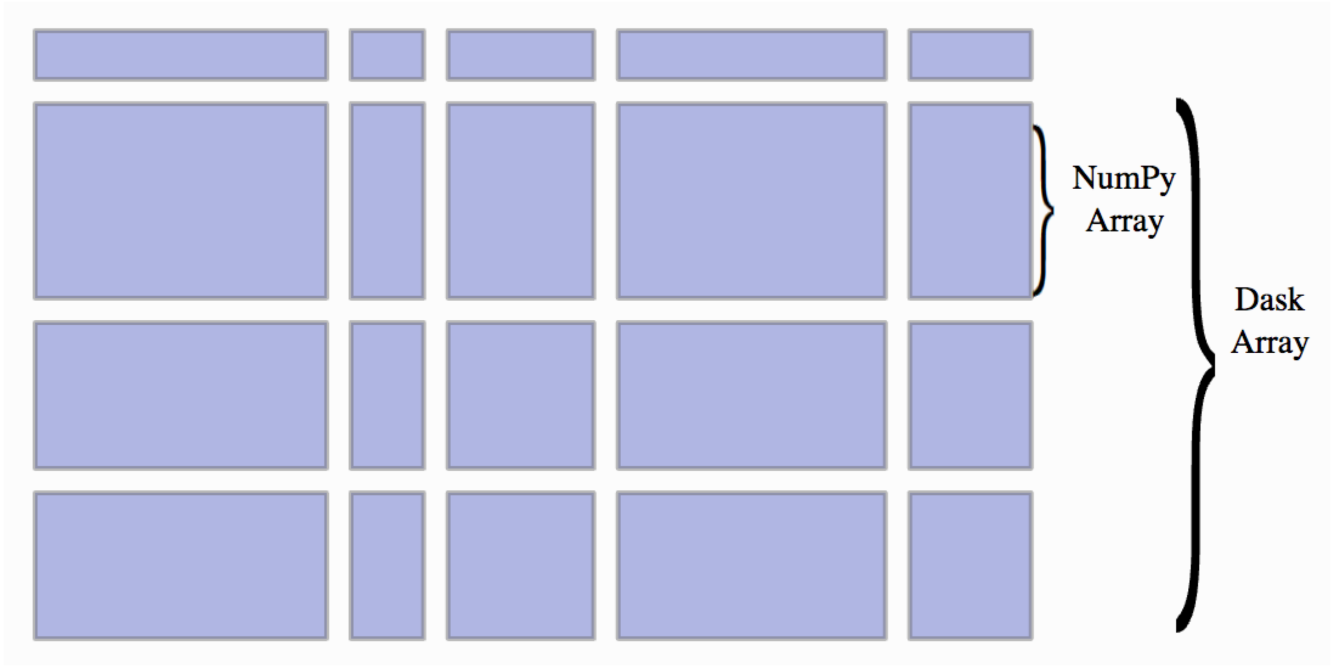
ACIDask built on top of ACIPython for connecting Dask applications to HPC libraries using Alchemist

- Sends data in Dask Arrays to Alchemist, and stores data sent from Alchemist in Dask Arrays
- Support for Dask DataFrames may be introduced in future

DASK Interface

Dask Arrays are actually a collection of smaller arrays (chunks) that fit in memory:

- Chunks are NumPy arrays or functions that produce arrays
- These arrays are arranged into a grid
- Dask Array coordinates their interaction with each other or other Dask arrays



ACIDask treats each chunk as a NumPy array and sends the data in each chunk to the appropriate Alchemist workers

PySpark Interface

PySpark is the Python API for Spark

- Built using popular Py4J library that allows Python to dynamically interface with JVM objects

ACIPySpark built on top of ACIPython for connecting PySpark applications to HPC libraries using Alchemist

- As with ACISpark, ACIPySpark supports RDD-based distributed data structures defined in MLlib's *linalg.distributed* module.

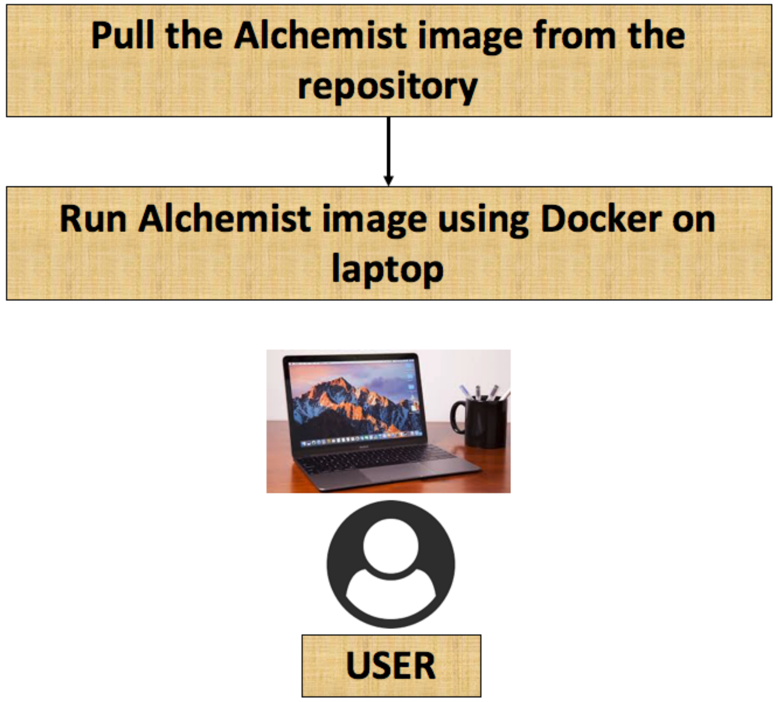
Deploying Alchemist on Different Platforms using Containers

Running Alchemist in a Container

- Ease of use on any platform that supports containers
- Flexibility of deployment across different platforms
- Relatively painful and time consuming to deploy natively
  - Elemental, ARPACK, Eigen, asio, spdlog and dependencies
  - Need to tweak install process if switching platforms
  - Difficult port limits availability to different kinds of users
- Common Dockerfile base:
  - Easily build and deploy Alchemist across several platforms using different container technologies (Shifter on XC, Singularity on CS, Kubernetes (local and cloud))
  - Some customization is necessary, e.g. building to target system-optimized MPI (need to support both MPICH and openMPI API)

Running Alchemist on Laptop using Docker

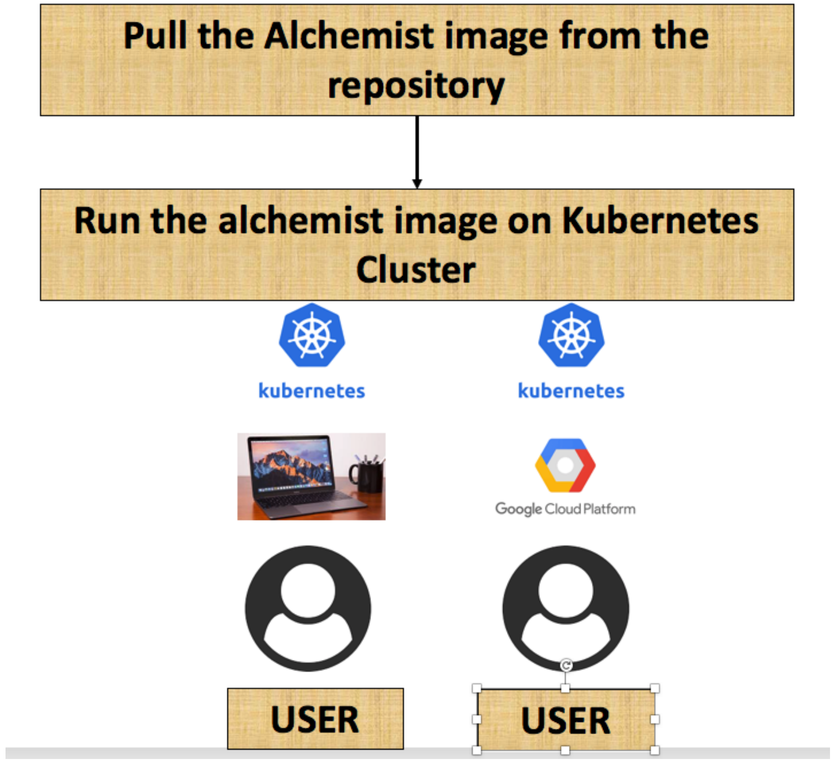
```
// Pull the image
docker pull projectalchemist/alchemist:latest

// Run Alchemist using docker
docker run -it --name alchemist -p  24960-
24963:24960-24963 \
    projectalchemist/alchemist:latest sh -c \
    "start_alchemist"

// Connect to alchemist using your favorite
interface: Spark/Python/Dask/PySpark
```


Pull the Alchemist image from the repository
Run Alchemist image using Docker on laptop
USER

Commands to run Alchemist on a Kubernetes Cluster

Run the Alchemist container in a Kubernetes Pod

```
// Pull the image
docker pull projectalchemist/alchemist:latest
// Create a Kubernetes namespace for Alchemist
kubectl create namespace alchemist
// Run Alchemist using the Docker image
kubectl run -it --namespace=alchemist alchemist-k8s
 --image=projectalchemist/alchemist:latest
 --port=24960 --port=24961 --port=24962 --port=24963
 -- /bin/bash -c "start_alchemist"
```


Pull the Alchemist image from the repository
Run the alchemist image on Kubernetes Cluster
kubernetes
kubernetes
Google Cloud Platform
USER
USER

Commands to run Alchemist on Cray Platforms

Initial Integration with Urika-XC/CS

- Leverage existing container launch scripts (run_training, start_analytics)
- Using a modified image which includes alchemist

```
// Grab an allocation from the existing cluster resource manager
 qsub -I -l nodes = 4
// Run alchemist using the run_training script
 run_training -v --no-node-list -n 4 \
 "$ALCHEMIST_EXE"
// Connect to alchemist using your favorite interface
Spark/Python/DASK/PySpark
```

Conclusions and Future Work

- We introduced new interfaces for Python, Dask and PySpark
- New Docker container allows for easy deployment, can be used with Shifter and Singularity on Cray systems, or with Kubernetes to run locally or on the cloud
- *Future work*:
    - Connect Alchemist with RLlib in Ray to enable reinforcement learning using HPC libraries
    - Support for ScaLAPACK-based HPC libraries
        - Currently Alchemist supports only Elemental-based HPC libraries

Github: project-alchemist/Alchemist

Introduced at CUG 2018 as

Alchemist: An Apache Spark ⇔ MPI Interface

Github: project-alchemist/Alchemist

Introduced at CUG 2018 as

~~**Alchemist: An Apache Spark ⇔ MPI Interface**~~

Now:

Alchemist: An HPC Interface for Data Analysis and Machine Learning Frameworks

Github: project-alchemist/Alchemist

THANK YOU

QUESTIONS?

kristyn@cray.com

SAFE HARBOR STATEMENT

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts.

These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.