Running Alchemist on Cray XC and CS Series Supercomputers: Dask and PySpark Interfaces, Deployment Options, and Data Transfer Times

Kai Rothauge
ICSI and Dept. of Statistics
UC Berkeley
Berkeley, CA, USA
kai.rothauge@berkeley.edu

Haripriya Ayyalasomayajula
Cray Inc.
Seattle, WA, USA
payyalasom@cray.com

Kristyn J. Maschhoff
Cray Inc.
Seattle, WA, USA
kristyn@cray.com

Michael Ringenburg
Cray Inc.
Seattle, WA, USA
mikeri@cray.com

Michael W. Mahoney
ICSI and Dept. of Statistics
UC Berkeley
Berkeley, CA, USA
mahoneymw@berkeley.edu

***Abstract*—Alchemist allows Apache Spark to achieve better performance by interfacing with HPC libraries for large-scale distributed computations. In this paper we highlight some recent developments in Alchemist that are of interest to Cray users and the scientific community in general. We discuss our experience porting Alchemist to container images and deploying it on Cray XC (using Shifter) and CS (using Singularity) series supercomputers, on a local Kubernetes cluster, and on the cloud.**

Newly developed interfaces for Python, Dask and PySpark enable the use of Alchemist with additional data analysis frameworks. We also briefly discuss the combination of Alchemist with RLlib, an increasingly popular library for reinforcement learning, and consider the benefits of leveraging HPC simulations in reinforcement learning. Finally, since data transfer between the client applications and Alchemist are the main overhead Alchemist encounters, we give a qualitative assessment of these transfer times with respect to different factors.

***Keywords*-Alchemist; Dask; PySpark; RLlib; MPI; Elemental; Cray XC; Cray CS; Shifter; Singularity; Kubernetes.**

I. Introduction

Alchemist [1], [2], [3] allows Apache Spark [4] to achieve better performance by interfacing with high-performance computing (HPC) libraries for large-scale distributed computations. The motivation for the development of Alchemist was the inadequate performance of distributed linear algebra operations in Spark's linear algebra and machine learning module, MLlib. It was found that not only are there significant overheads when performing the operations in Spark up to an order of magnitude greater than the actual execution time of the distributed operation but these overheads in fact anti-scale, i.e. they increase faster than the execution time of the operation as the data sets increase in size.

Alchemist was designed to alleviate this problem by allowing users to easily interface with existing or custom HPC libraries. Efficiently implemented MPI-based linear algebra libraries do not suffer from the anti-scaling behaviour of MLlib or from large overheads not related to the execution of the actual linear algebra operation, but are generally difficult to use for practitioners not familiar with them or with HPC in general. Alchemist therefore combines the best of both worlds: the high productivity of Spark, allowing users to make use of its numerous data analysis components, and the high productivity of HPC libraries that can perform large-scale distributed operations faster than Spark can.

After giving a brief overview of the Alchemist framework in Section II, we discuss some of Alchemist's recent developments that are of interest to Cray users and the scientific community in general. Alchemist is no longer an HPC interface just for Spark and can, in principle, be used by any data analytics framework, given a suitable client interface, and in Section III we introduce new client interfaces for Python, Dask and PySpark. Alchemist can also be used in applications other than data analysis and we briefly discuss the potentially exciting combination of Alchemist with a reinforcement learning framework in Section IV, although a detailed case study will be the subject of future research. In Section V we discuss the deployment of Alchemist on different platforms using recently developed container images. While Alchemist does not suffer from the overheads that are incurred by Spark, some overheads are encountered when transmitting the data sets from the client application to Alchemist, and in Section VI we try to quantify these transfer times by taking various factors into account, namely matrix layouts, message buffer sizes, and variability in the network communication times due to varying network loads.

II. Overview of Alchemist

Here we briefly review Alchemist—for a more extensive discussion, see [1], [2]. The basic framework of Alchemist is given in Figure 1: a client application (which is a Spark

Figure 1. Overview of the basic Alchemist framework

application in the figure) connects to Alchemist using a suitable Alchemist-Client Interface (ACI). All communication between the client application and Alchemist occurs through the ACI. The client interface requests a number of workers from Alchemist and each of its executors connects to each of the Alchemist workers. The client interface can specify which HPC libraries it wishes to use, and these libraries are loaded by the connected Alchemist workers dynamically. Each HPC library requires a corresponding Alchemist-Library Interface (ALI) that imports the HPC library and provides wrapper functions for every function in the HPC library that is of interest. It also provides a standard interface for Alchemist and calls the desired function(s) in the HPC library in the required format.

Communication between the client interface and Alchemist is primarily between the client driver process and the Alchemist driver process. If distributed data sets need to be transferred between the client interface and Alchemist, then this is done between the client workers and the Alchemist workers, where each client worker sends its portion of the data to the connected Alchemist workers. These data sets will be in the form distributed matrices that require some method of storing them, and to this end Alchemist makes use of the Elemental [5] library. Elemental is an MPI-based library that provides a convenient interface for storing distributed matrices (called `DistMatrices`), although using Elemental comes at the cost of requiring that the HPC libraries use Elemental as well so that they can access the data in the `DistMatrices`. Alchemist will also provide support for ScaLAPACK in a future version.

III. Python, Dask, and PySpark Interfaces

As mentioned above, Alchemist was originally written as an interface between Scala-based Apache Spark and MPI-based libraries, but recent extensions have allowed client interfaces for other languages and data analysis frameworks to be easily developed. To this end, a Python [3] interface has been written and serves as a basis for client interfaces for Dask [6], a popular library that supports parallel computing in Python, and PySpark.

The ability to use Alchemist from these additional frameworks enables more users to easily connect to HPC libraries. We will describe each of these interfaces in turn.

A. ACIPython: Alchemist-Client Interface for Python

Python has established itself as the most popular language for data analysis and machine learning tasks, therefore substantial effort has been spent on the development of a client interface for Python. This allows Python users to connect to Alchemist and make use of existing HPC libraries for their data analysis and machine learning needs. We note that while there already are Python bindings for MPI (for instance the MPI4Py library [7]) that allow a Python program to exploit multiple processors, our purpose is different in that we allow users to easily connect to existing or custom HPC libraries, in particular when Alchemist is running remotely. The Python interface does not require the installation of any additional packages aside from ACIPython, and does not require the installation of Alchemist, MPI, or any HPC libraries if connecting to Alchemist remotely or when running Alchemist from inside a container.

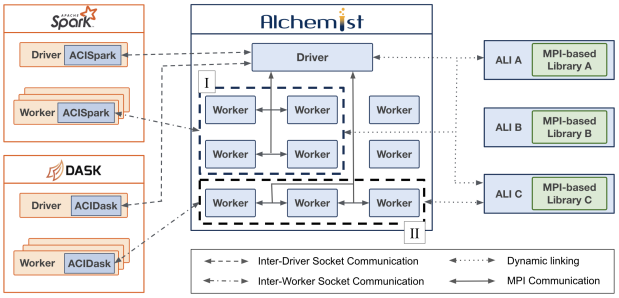

Figure 2. An illustration of Alchemist in use: A Spark application connects to Alchemist and requests 4 workers, which Alchemist provides by creating a group of workers that the Spark application can connect to. The Spark application wishes to use functions in Libraries A and C, so the Alchemist workers allocated to the Spark application load these libraries dynamically. Distributed data sets are transferred between the Spark and Alchemist workers. At the same time, a Dask application connects to Alchemist and requests 3 workers, which Alchemist provides, as well as access to the requested Library C.

The design of ACIPython resembles that of the Spark interface. As described in Section 2, the user connects to Alchemist via the Python interface and requests a certain number of workers. Communication is primarily with the Alchemist driver, but large matrices (or other large data sets) are sent to the Alchemist workers.

An important difference is that the Python interface assumes that the underlying application is running on a single process, so that all data that is sent to Alchemist is small enough to fit in memory of the machine that the application is running on. This means that at this point Python applications running on multiple processors, for instance using MPI4Py, are not yet supported in general, although see the ACIDask and ACIPySpark interfaces described below. The reader may question the usefulness of using Alchemist with data that is small enough to fit on a single machine, but there are several scenarios that come to mind:

- If the data can be loaded from a file that is accessible to Alchemist, it can be loaded by Alchemist directly and there is no need for the client application to load the data and transfer it. In this case it does not matter

```
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.

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
  ------------------------------------------------------------------------------------------------------------

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]]
```

Figure 3. Screenshot of a Jupyter notebook in which Alchemist is called using ACIPython. In this simple example the Python application connects to Alchemist, requests access to 3 workers, and loads the test library TestLib (a simple MPI-based library that provides a small set of test functions, including the truncated SVD). A randomly generated NumPy array of size $1,000 \times 1,000$ is sent to the Alchemist workers, which then perform the rank-10 decomposition of it. Alchemist returns handles to each of the output matrices (the 10 left singular vectors in U, the 10 right singular vectors in V, and the singular values on the diagonal of S, but in this case we are interested only in the singular values and we use `fetch_matrix` to receive the entries of S from Alchemist.

if the data is too large to fit on a single machine, as long as Alchemist has allocated a sufficient number of worker nodes.

- The client application may also load or generate data that is too large to fit in memory in chunks, and then transfer each chunk to Alchemist. In this way large data sets that are too large to fit in the memory of a single machine can be transmitted.
- Some computations may 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 be significantly smaller and fit easily inside the memory of a single machine.

These are some of the circumstances under which Alchemist and its Python interface make it easy to access HPC libraries from a Python application, even if the Python application itself is not designed for parallel computations. The Python interface also serves as the basis for Alchemist interfaces that do run on multiple processes, for instance the Dask and PySpark interfaces described below.

ACIPython assumes that all data sets of interest can be represented by, or converted to, NumPy arrays. The data in the array, or a subset of it, is then serialized and sent to each of the connected Alchemist workers sequentially, where they are stored in an Elemental `DistMatrix`. Each Alchemist worker receives a different chunk of the data;

for instance, when transferring a $10,000 \times 10,000$ array to 10 Alchemist workers using a row-major layout (see Section VI), each of the workers will receive every 10th row of the array. Transmitting data from Alchemist back to the Python application is similarly straightforward, in this case the data, or a subset of it, in an Elemental `DistMatrix` is transmitted from Alchemist to the client application, where it is deserialized and stored in a NumPy array.

See the screenshot of the Jupyter notebook shown in Figure 3 for an illustration of the use of ACIPython.

B. ACIDask: Alchemist-Client Interface for Dask

Dask is a popular scalable data analytics platform for Python that is designed to integrate with existing applications. It provides data structures such as arrays and dataframes for storing data in larger-than-memory or distributed environments, and these parallel collections run on top of dynamic task schedulers that are optimized for computation.

ACIDask provides a convenient interface, built on top of ACIPython, for connecting Dask applications to HPC libraries using Alchemist. Our primary interest is in transmitting data stored in a Dask array to Alchemist, where it is then accessible to HPC libraries. Dask arrays are used in fields like atmospheric and oceanographic science, genomics, numerical algorithms for optimization or statistics, large scale imaging, and more, and all of these applications can potentially benefit from access to general-purpose or domain-specific HPC libraries.

Dask arrays are actually a collection of many smaller arrays, referred to as *chunks* or *blocks*, that may be NumPy arrays or functions that produce arrays; if they are actual arrays, they may be stored on disk or on other machines. These arrays are arranged into a grid and the Dask array coordinates their interaction with each other or other Dask arrays. Dask arrays implement a subset of the NumPy ndarray interface using blocked algorithms.

The approach taken by ACIDask is to work with the individual chunks that compose the Dask array and send them to an Elemental `DistMatrix`. Each Dask array `x` has a unique name that can be accessed using `x.name`, and every chunk in the array is referred to by the tuple `(x.name, i, j)`, with `i`, `j` being the indices of the block ranging from 0 to the number of blocks in that dimension[1]. The (i, j)th chunk can be accessed by the code shown in Figure **??**. In both cases `x_ij` is a NumPy array containing the data of the (i, j)th chunk, which ACIDask then sends from the Dask process storing the chunk to Alchemist.

[1]Dask actually accepts up to three indices `i`, `j`, `k`, and can therefore store 3-dimensional arrays, not just matrices. Since Elemental does not support higher-dimensional arrays, we restrict ourselves to Dask arrays representing vectors or matrices

```
# Extract the (i,j)—th chunk from a Dask Array A
def get_chunk(A, i, j):
    layers = A.dask.layers[x.name]
    a = layers[(A.name, i, j)]

    # Chunks are functions that produce NumPy arrays
    return a[0](*a[1])
    # OR
    # Chunks are actual NumPy arrays
    return a[0](layers[a[1]], a[2])
```

If the function in the HPC library returns a distributed matrix, Alchemist sends the dimensions of the matrix back to ACIDask, which then builds a Dask array that can store the data. Each Dask process then requests the data corresponding to its chunk from Alchemist and inserts it into the Dask array.

Support for Dask dataframes and other constructs may be introduced in future.

C. ACIPySpark: Alchemist-Client Interface for PySpark

Given that the original purpose of Alchemist was to accelerate and extend the functionality of Apache Spark when working with large, distributed data sets, it is only natural to extend the Python interface to support PySpark, the Python API for Spark, built using the popular Py4J library that is integrated within PySpark and allows Python to dynamically interface with JVM objects. Python generally offers improved readability of code and ease of use and maintenance compared to Scala, and PySpark has therefore become a popular interface for working with Spark's various features and libraries. For users wishing to use Spark with Alchemist, but reluctant to work with Scala, we recommend using PySpark with ACIPySpark.

As with ACISpark, ACIPySpark supports RDD-based distributed data structures defined in MLlib's `linalg.distributed` module. In particular, ACIPySpark supports `BlockMatrix`, *CoordinateMatrix*, `RowMatrix`, and `IndexedRowMatrix`, which represent distributively stored matrices backed by one or more RDDs derived from `DistributedMatrix`[2].

ACIPySpark does not first convert local submatrices of a distributed matrix in PySpark into NumPy arrays before sending the data over to Alchemist. Instead, the data from the `DistributedMatrix` is serialized directly into the message buffer. Likewise, if the HPC library returns a distributed matrix, Alchemist sends the dimensions of the matrix back to ACIPySpark, which then builds a `DistributedMatrix` array to store it. Each PySpark process then requests the data corresponding to its local

[2]As of Spark 2.0, Spark is moving to a Dataframe-based API in the `spark.ml` package for its linear algebra and machine learning operations. Support for DataFrames will be introduced in future versions of ACISpark and ACIPySpark

submatrix from Alchemist and inserts the deserialized entries into the `DistributedMatrix`.

IV. RLlib + Alchemist for Reinforcement Learning with HPC Simulations

Reinforcement learning (RL) [8] is an exciting area of machine learning that allows a (simulated) learner to learn by interacting with a simulated environment via a series of rewards, with the goal being to maximize the number of accumulated rewards by the end of the training. The learner must find which actions to take to obtain the maximum number of rewards independently, and therefore, due to its trial-and-error approach, a large number of simulations are required in order to successfully train the learner. While the computational cost of these simulations may be unimportant when applying RL to small problems that are commonly used to illustrate its usefulness, it becomes a significant bottleneck when applying RL to large-scale problems in science and engineering that require appreciable computational resources.

It is therefore of interest to enable reinforcement learning packages to call HPC libraries for the simulations. There are potentially many areas in science and engineering that would benefit from this, in particular areas which traditionally require expensive HPC simulations and where some set of constraints and optimality conditions has to be met (airplane design, drug discovery, etc.). The rewards given to the learner reflect how well the current set of parameters satisfies these criteria.

RLlib [9] is an open-source library for RL that is based on the Ray [10] framework. It provides a collection of RL algorithms and scalable primitives for composing new ones. It has seen a significant increase in interest recently, and a compelling use case of Alchemist's Python interface is in providing a simple interface through which the user of RLlib can call HPC libraries for the simulations. Alchemist thereby allows users to employ efficient HPC libraries for the simulations while still working with the extensive tool set and convenient interface provided by RLlib, hopefully facilitating the adoption of RL by the scientific and engineering communities.

A detailed case study will be the subject of future work. Here we simply illustrate how one could call an HPC library through Alchemist inside a Python script given the current RLlib API. RLlib makes use of OpenAI Gym, a toolkit for developing and comparing RL algorithms; we omit a lot of the details since these will be discussed at a later point in time.

First, we need to create the class in which the simulation environment is defined:

```
class HPCSimulator(gym.Env):
  # Initialize simulation environment
  def __init__(self, config):
    hostname    = config["hostname"]
    port        = config["port"]
    num_workers = config["num_workers"]
    lname       = config["lib_name"]
    lpath       = config["lib_path"]

    self.als = AlchemistSession()
    self.als.connect(hostname, port)
    self.als.request_workers(num_workers)
    self.HPClib = self.als.load_library(lname, lpath)

  # Reset simulator
  def reset(self):
    self.HPClib.reset()
    return self.HPClib.get_state()

  # Take a step in the simulation in response
  # to an action
  def step(self, action):
    self.HPClib.step(action)
    return self.HPClib.get_state(), self.HPClib.get_score()
```

In the above sample listing, the `HPCSimulator` class is derived from OpenAI Gym's Environment class. An `AlchemistSession` is set up during initialization, and in this case we have opted that all pertinent settings are contained in a dictionary (which we called `config` here), although of course one could also read them from file. As before, we need to connect to Alchemist, request a certain number of workers, and get Alchemist to load the HPC library we want to use, denoted by `HPClib`. Presumably `HPClib` has an efficient simulator implemented that we want to use during our training procedure. To run with RLlib, `HPClib` needs to define `reset`, to set the simulator's state to its default configuration; `step`, to advance the simulation by one step in response to the `action`; `get_state`, to return the simulators current state; and `get_score`, to evaluate how well the current state does with regard to some problem-specific optimality condition.

To use the simulator with RLlib, we simply provide the class name as the environment within Tune, which is Ray's scalable hyperparameter search framework (a discussion of Tune lies outside the scope of this paper). For example:

```
if __name__ == "__main__":
  ray.init()
  ModelCatalog.register_custom_model(...
                        "my_model", CustomModel)
  tune.run(
    "PPO",
    stop={"timesteps_total": 10000,},
    config={
      "env": HPCSimulator,
      # more configuration options ...
```

```
    }
  }
)
```

See the documentation for Ray, RLlib and Tune for a clearer understanding of their APIs. The sample listings given here are just to give a flavor of what the combination of RLlib with Alchemist might look like, actual implementations may vary.

V. Deploying Alchemist on Different Platforms using Containers

In this section we discuss our experiences porting Alchemist to container images and deploying it on Cray XC (using Shifter) and CS (using Singularity) series supercomputers, on a local Kubernetes cluster, and on the cloud.

Container images allow us to bundle applications and their dependencies together into a single entity. We deploy the Alchemist image on the host machine to run Alchemist in a container and the client application can connect to it. Moving to a container-based deployment means users do not need to worry about building the applications from source and managing dependencies every time they want to run their application on a new platform. We will discuss deploying Alchemist using four major container technologies: Docker, Singularity, Shifter, and Kubernetes. Docker is an open source container technology that has gained wide adoption. We start our discussion by describing how we package Alchemist into a Dockerfile, a construct that lets users define software specification applications and their runtime environment. As an example, we will demonstrate running the Docker image on a laptop, which is usually the initial development environment from most researchers. As an example of how moving to Docker solves the problem of consistently building the dependencies, we discuss building the Elemental library with different versions of gcc, gcc++, gfortran and MPI.

Cray XC series supercomputers use Shifter, developed at NERSC to deploy container images. We highlight how we can reuse the Dockerfile we developed earlier, by making minor changes to deploy Alchemist on Cray XC. The changes allow us to leverage the Cray MPI library stack. Cray CS series supercomputers use Singularity for launching container images on the nodes, which provides flexibility to import Docker images without having Docker installed or being a superuser. We leverage this ability to run Alchemist on CS systems with optimized OpenMPI libraries.

Kubernetes is an open source orchestration framework that has gained popularity both in cloud and on-premise clusters. It supports running, scaling and management of containers. We use the Alchemist Docker image to run Alchemist on Kubernetes and we demonstrate running this on a local Kubernetes cluster. Users can follow the same steps they used on a local kubernetes installation to run alchemist on kubernetes cluster deployed on the Google Cloud Platform. While there are subtle differences between each of these flavors of container orchestration, it provides the flexibility for users to choose their favorite container technology as opposed to being restricted to one single mode of deployment.

```
// 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 /bin/bash -c \
    "start_alchemist"
```

Figure 4. Commands to build the Alchemist Docker image and run them Alchemist using Docker on a laptop

The containers discussed here are available on [3].

A. The Alchemist Docker Image

The first step to containerizing Alchemist involves writing a Dockerfile. Dockerfile is a configuration file with commands to install a base operating system followed by different software components and dependency libraries in an image. It provides us with a clean slate where we can customize an operating system of our choice followed by different software components that are needed for our application. Once we have the Dockerfile in place, we build it using the Docker build command and push it to the Docker registry.

For our alchemist Dockerfile, we use the latest version of Debian operating system. The Dockerfile includes commands to install necessary compilers and other libraries followed by commands to install the required dependencies such as Elemental, SPDLog, asio and finally commands to install Alchemist. The recipe to build the Alchemist Docker image is listed in Figure 4. Figure 5 shows a screenshot of running Alchemist using Docker on a laptop.

Once we have Alchemist running as shown in the screenshot above, we can connect to it from a client application, assuming the client application has imported the appropriate client interface.

Without containers it would take a significant amount of work to download and install Alchemist with all of its dependencies, a tedious and time consuming process, but with the Docker image the users can instead focus on their workflow.

B. Deploying Alchemist Images on Urika-XC using Shifter and on Urika-CS systems using Singularity

In this section, we describe our initial efforts to run Alchemist on Cray XC series supercomputers using Shifter containers. We re-used the Dockerfile from running Alchemist using Docker on local machine. We integrate this with the existing container launching scripts and build the Shifter container.

```
C02V25U2HTD6:~ payyalasom$ docker run -it --name alchemist -p  24960-24963:24960-24963 projectalchemist/alchemist:latest /bin/bash -c "
start_alchemist"
/data/Alchemist-main
/data/Alchemist-main/target/alchemist
[2019-05-07 02:29:25.576] [driver] [info]        ============================
                                                 Starting Alchemist 0.5
                                                 ----------------------------
                                                 Running on dbeec2290000 0.0.0.0:24960
[2019-05-07 02:29:25.579] [driver] [info]        Starting workers
[2019-05-07 02:29:25.579] [driver] [info]        Sending command START to workers
[2019-05-07 02:29:25.976] [worker-001] [info]    Worker ready
                                                 Running on dbeec2290000 0.0.0.0:24961
[2019-05-07 02:29:25.976] [worker-002] [info]    Worker ready
                                                 Running on dbeec2290000 0.0.0.0:24962
[2019-05-07 02:29:25.976] [worker-003] [info]    Worker ready
                                                 Running on dbeec2290000 0.0.0.0:24963
[2019-05-07 02:29:25.976] [driver] [info]        Registering workers
[2019-05-07 02:29:25.976] [driver] [info]        Sending command SEND INFO to workers
[2019-05-07 02:29:26.378] [worker-003] [info]    Sending hostname and port to driver
[2019-05-07 02:29:26.378] [worker-001] [info]    Sending hostname and port to driver
[2019-05-07 02:29:26.378] [worker-002] [info]    Sending hostname and port to driver
[2019-05-07 02:29:26.378] [driver] [info]        3 workers ready
[2019-05-07 02:29:26.378] [driver] [info]        List of workers (3):
                                                     Worker-001 running on dbeec2290000 at 0.0.0.0:24961 - IDLE
                                                     Worker-002 running on dbeec2290000 at 0.0.0.0:24962 - IDLE
                                                     Worker-003 running on dbeec2290000 at 0.0.0.0:24963 - IDLE

[2019-05-07 02:29:26.378] [driver] [info]        ============================
                                                 Alchemist is ready
                                                 ----------------------------
[2019-05-07 02:29:26.378] [driver] [info]        Accepting connections ...
```

Figure 5. Screenshot showing Alchemist running on a laptop using Docker

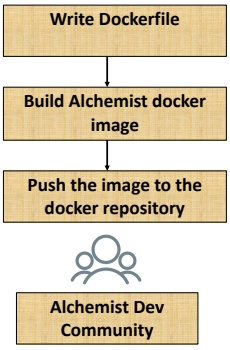

Figure 6. Workflow: Building Alchemist container image

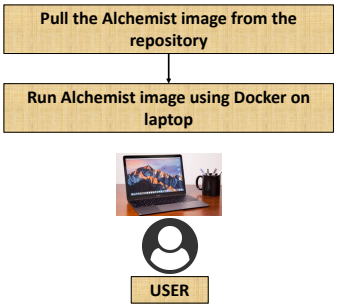

Figure 7. Running Alchemist image using Docker on a laptop

We use Singularity to run Alchemist images on Cray CS series super computers. We continue to use the base Dockerfile from running Alchemist using Docker on a laptop. We build this using Open MPI libraries that are unique to CS series super computers.

Whether it is XC or CS, to provide a uniform experience, we run alchemist using the run_training script that is a part of the existing container launch scripts. The commands to

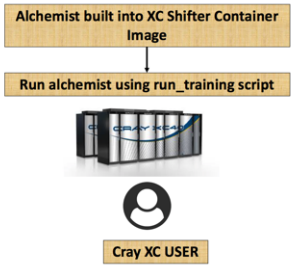

Figure 8. Running Alchemist using Shifter on Cray XC systems

```
// Load the analytics module:
module load analytics

// Grab an allocation from the existing
// cluster resource manager SLURM/PBS:
qsub -I -l nodes = 4

// Run alchemist using the run\_training script
run_training -v --no-node-list -n 4 \
"/data/Alchemist-main/target/alchemist"
```

Figure 9. Commands to run Alchemist on Cray systems

run alchemist on XC and CS systems are listed in 9

C. Deploying the Alchemist Docker Image on a Kubernetes Cluster

Kubernetes, an open source orchestration framework, gained wide adoption over the last few years and deploying Alchemist on a Kubernetes cluster makes it available to different user communities. It helps enterprise users to experiment with great ease without having to worry about the internals of setting up and configuring Alchemist and its dependencies.

We use the Docker image we created in the first step to

```
// 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"

// Expose the kubernetes deployment
kubectl expose deployment alchemist-k8s -n alchemist \
 --port=24960 --port=24961 --port=24962 --port=24963 --name=alchemist

// Get the podid
kubectl get pods -n alchemist

//Set up port forwarding
kubectl port-forward alchemist-k8s-<podid> -n alchemist \
 24960:24960 24961:24961 24962:24962 24963:24963
```

Figure 10. Commands to run Alchemist on a Kubernetes cluster

deploy Alchemist on a Kubernetes cluster. Whether it is a local Kubernetes cluster or a cluster configured on a cloud platform, we can use the same commands to run Alchemist. We start by creating a Kubernetes *namespace*, which is the abstraction Kubernetes uses that provides isolation to different users in a cluster. We can have multiple groups in an organization connect to different instances of the same Kubernetes cluster using different namespaces. We then use the same Docker image and create a Kubernetes deployment for Alchemist. We will instantly see Alchemist running in a Kubernetes *pod*, the basic building block of Kubernetes. Pods are the smallest and simplest units in the Kubernetes object model that can be created or deployed and represent a running process in the cluster. A pod can run any number of containers. There are two stages involved in running Alchemist on a Kubernetes cluster: run the container on the Kubernetes cluster, then expose the ports by setting up port-forwarding to be able to connect to Alchemist from a client interface.

The commands to run alchemist on Kubernetes cluster are listed in Figure 9.

VI. Evaluating Communication Overheads

As discussed in [1], the main computational overhead of Alchemist is the time it takes to transfer the data between the Spark application and the HPC libraries. A simple experiment to quantify these communication times for two 400GB matrices with different shapes was performed (see Tables 2 and 3 in that reference). It was observed that there is significant variability in the communication times, governed by two major factors: the *number of messages* sent across the network and the *transfer time variability*.

A. Factors impacting communication times

The variability of the transfer times stems mainly from variable network loads. It will generally take longer to transmit a large amount of data if the network is in heavy use, but it may also be the case that the communication between only a small number of nodes is impacted, which will still lead to a higher overall transfer time if some of the data has to be sent between these nodes. In general, we expect that a larger number of small messages will have more variability compared to a small number of large messages, simply due to the increased likelihood that some of the messages will be delayed at some point while being transmitted across the network. Since the simulations cannot proceed until all of the data has been transferred, even one straggler can cause a higher measured transfer time.

On the other hand, it is generally more efficient for sockets to handle smaller messages, and larger messages may in fact lead to network blockages. Also, a large number of small messages sent between a large number of nodes means that more of the data is sent concurrently and one would therefore expect, under optimal conditions, smaller transfer times.

There are several (not necessarily independent) factors that influence the number of messages sent across the network:

- *Amount of data*: The amount of data that needs to be sent across the network is determined by both the size of the matrix and the size of its datatypes in memory (for instance `doubles` vs. `floats`).
- *Message buffer size*: Larger buffers allow for fewer messages, but having large messages may have adverse effects, such as taking up too much memory on a core (leaving less for the actual data), and causing network blockages.
- *Number of Alchemist processes*: A larger number of Alchemist workers may accelerate certain computations, but it comes at the price of an increased number of messages, both between the workers during the computation, and (more importantly for us) between Alchemist and its client interface. This is counterbalanced by the messages being shorter and more communication happening concurrently.
- *Number of Spark partitions*: Apache Spark divides its RDDs into a number of partitions, and all tasks are then performed on these partitions in parallel, including sending the data over to Alchemist. The exact number of partitions that Spark uses depends on several factors, but generally one would expect to have at least one partition per core. With a large number of cores, this would mean that one would have a significant number of partitions that all need to connect to Alchemist concurrently to send their data, and leading to a large number of small messages being sent. Since the data of a lot of these partitions is physically located on the same nodes, one would hope to be able to combine the data from several of the partitions destined for the same Alchemist worker before sending it across the network, but this is impossible given Spark's current API. The drawback of having each partition com-

municate with Alchemist directly is the number of network connections that have to be opened between the Spark application and Alchemist; even with only a dozen nodes allocated to the Spark application and Alchemist respectively, the number of partitions will be in the hundreds if there are a lot of cores on each node. The number of network connections will be in the thousands, and opening each of them incurs an overhead that may dominate the time it takes to send and receive the actual data if the messages are small. The rule of thumb here is that one wants to have the data in the Spark application be in the smallest number of partitions possible, i.e. each partition should hold has much data as possible, to minimize the communication overheads. This means not allocating more nodes to the Spark application than needed.

- *Matrix layout*: The layout used by the Elemental `DistMatrices` can have a significant impact on the performance of the HPC libraries, so it may be desirable to send the data from Spark to a `DistMatrix` that has a more favorable layout for the computations that are going to be performed on it. However, some layouts may require more messages to be sent across the network than others, for instance if a particular layout requires the local entries on one Alchemist worker to be sent from a large number of Spark partitions vs. a small number.
- *Aspect ratio of the matrix*: The aspect ratio of the matrix (its height-to-width ratio) will have an impact as well, as was found in the previous study, where sending the rows of an `IndexedRowMatrix` is more efficient and less variable if the matrix is short and wide rather than tall and thin. This is due to a smaller number of larger messages being sent, with the messages not being large enough to adversely affect communication across the network. Due to the structure of `IndexedRowMatrices` and the row-based layout used by the Elemental `DistMatrix` in that study, it also means that the partitions needed to send data to fewer Alchemist workers, leading to fewer network connections having to be opened.

We do not discuss here the time it takes to serialize and deserialize the data, but this of course has an impact on the communication times as well. Recent improvements in Alchemist and its client interfaces have managed to decrease this overhead significantly.

Instead, here we are concerned with understanding the impact of the matrix layouts on the transmission times, but we also consider the message buffer sizes. A comprehensive study of the combined effect of all of the above factors lies outside the scope of this paper, but may be performed in future work.

B. `DistMatrix` layouts

See [11] for a discussion of different matrix layouts in Elemental that are possible with respect to the *process grid*. The process grid is Elemental's two-dimensional arrangement of the worker processes associated with a given `DistMatrix`. For simplicity, let us assume that there are 6 workers with IDs $1, \ldots, 6$ that Elemental has arranged in a 2×3 process grid P:

$$P = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}.$$

There are several distribution schemes that Elemental defines, which we list here:

- CIRC: Only give the data to a single process;
- STAR: Give the data to every process;
- MC: Distribute round-robin within each column of the 2D process grid (*M*atrix *C*olumn);
- MR: Distribute round-robin within each row of the 2D process grid (*M*atrix *R*ow);
- VC: Distribute round-robin within a column-major ordering of the entire 2D process grid (*V*ector *C*olumn);
- VR: Distribute round-robin within a row-major ordering of the entire 2D process grid (*V*ector *R*ow);
- MD: Distribute round-robin over a diagonal of the tiling of the 2D process grid (*M*atrix *D*iagonal).

The layout of a `DistMatrix` is defined by one of thirteen different legal distribution pairs `(colDist,rowDist)`. Some of these layouts allow for data to be stored redundantly (i.e. the same matrix element may be on multiple processes), and these layouts are not important for our purposes. We illustrate the layouts in Elemental that do not store the data redundantly for a sample 7×7 matrix. The entries in the matrix correspond to the ID of the worker that that particular entry in the matrix is stored on.

- [MC, MR]: The majority of parallel routines in Elemental expect the matrices to have this layout, but it may not be the optimal layout for all purposes. Note that the process grid is tessellated with this distribution pair.

$$\begin{bmatrix} 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 2 & 4 & 6 & 2 & 4 & 6 & 2 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 2 & 4 & 6 & 2 & 4 & 6 & 2 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 2 & 4 & 6 & 2 & 4 & 6 & 2 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \end{bmatrix}$$

- [MR, MC]: Note that the transpose of the process grid

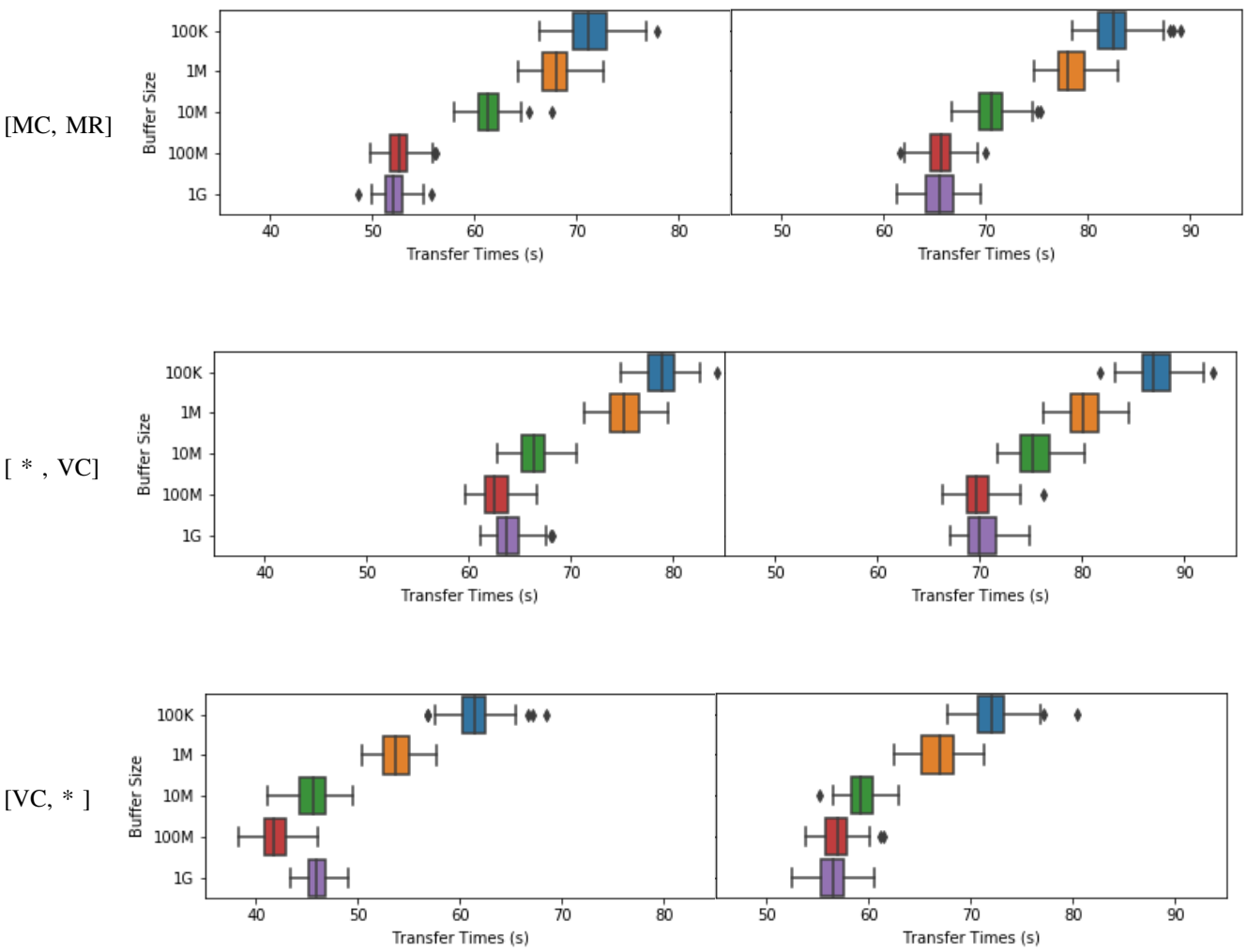

Figure 11. Data transmission times for various matrix layout and message buffer sizes. On left we have results for an `IndexedRowMatrix` of size $250,000 \times 200,000$, and on the right an `IndexedRowMatrix` of size $1,000,000 \times 50,000$. The matrix was sent from the Spark application to Alchemist 50 times, with the transfer times represented by the box plots. Note that the transfer times decrease and become less variable as the message buffer sizes increase. Actual transfer times are subject to change with further development of Alchemist's serialization protocol, we are interested in the general trends shown by these plots.

is tessellated with this distribution pair.

- [STAR, VC]:

$$\begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 2 & 1 \\ 3 & 4 & 3 & 4 & 3 & 4 & 3 \\ 5 & 6 & 5 & 6 & 5 & 6 & 5 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 \\ 3 & 4 & 3 & 4 & 3 & 4 & 3 \\ 5 & 6 & 5 & 6 & 5 & 6 & 5 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 & 1 \end{bmatrix}$$

- [VC, STAR]:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Not shown here are the layouts [VR, STAR] and [MD, STAR], which are similar to [VC, STAR] but with the rows permuted. Likewise, we do not consider [STAR, VR] and [STAR, MD], since these are similar to [STAR, VC] but with the columns permuted.

Some of these layouts may not be appropriate for all cases, for instance it may not be possible to store entire rows or columns on a single process if the matrices are too wide or tall, respectively.

C. Experiment

We run our experiments on Cori [12], a Cray XC40 supercomputer administered by NERSC. We use its Intel Xeon "Haswell" processor nodes, each of which have 32 cores and 128GB of memory. Nodes on Cori are communicated using the Cray Aries interconnect.

For our experiment we send a 400GB `IndexedRowMatrix` of `doubles` to an Elemental `DistMatrix` of the same dimensions. We look at the effect of the above layouts on the communication times and also take different message buffer sizes into account. For brevity of exposition, we only consider two different matrix dimensions: $250,000 \times 200,000$ and $1,000,000 \times 50,000$. For a given layout and buffer size, the matrix is sent to Alchemist 50 times at intervals of 30 minutes in order quantify the variability of transmission times due to network loads over a stretch of time.

On Cori, all software is managed using a modules software environment, which we use to load Spark 2.3.0. Alchemist and its dependencies are compiled from scratch and run natively on Cori. It was found that Spark has difficulties communicating with Alchemist when running within the same job, therefore we instead run Spark and Alchemist as separate jobs concurrently, with the user connecting the Spark application to Alchemist by providing it with the hostname of the node that the Alchemist driver is running on (one should therefore start the Alchemist job before the Spark job if not running in interactive mode). For the purposes of this experiment, we run the Spark application on four nodes, and allocate five nodes to Alchemist—one for the driver, four for the workers that will actually store the data. Since we have four Alchemist workers, the process grid will be square and there is no appreciable difference between the [MC, MR] and [MR, MC] distributions, therefore we do not consider the [MR, MC] distribution in this experiment.

The results of the experiment are shown in Figure 11. We report the communication times from Spark to Alchemist for the $250,000 \times 200,000$ matrix on the left, and the $1,000,000 \times 50,000$ on the right; communication times from Alchemist to Spark are similar, so we do not report them here.

In general one can conclude that it is better to have larger message buffers rather than smaller ones, but only up to a point, with 100MB seemingly a good compromise. It is generally faster to send matrices that are wider rather than narrower, although this is an artifact of `IndexedRowMatrices` storing data in rows. This also explains why sending data to Alchemist is faster if the `DistMatrix` uses a [VC, STAR] layout, since Spark is sending the data from rows to rows. In contrast, a [STAR, VC] layout requires the data in rows to be sent across columns that may be stored on different nodes by the `DistMatrix`, resulting in significantly more messages with less data and thereby increasing the overall communication times. The [MC, MR] layout is slightly more expensive than the [VC, STAR] layout since it again requires more messages to be sent, but most distributed operations will perform faster with this layout and it is expected that it is worth the additional communication cost; the same may apply to the [STAR, VC] layout in the right context.

Note that the communication times are subject to change as Alchemist's serialization protocol continues to improve. We are interested here in the general trends shown by the communication times, not the actual times themselves.

VII. Summary

Several recent developments have enabled more practitioners to use Alchemist to easily access HPC libraries from data analysis frameworks such as Spark, Dask and PySpark, or from single-process Python applications. The availability of Docker and other containers enables users to get started with Alchemist quickly, and we briefly discussed the potentially exciting combination of Alchemist with reinforcement learning frameworks such as RLlib. Alchemist's main overhead comes from the data transfer between client applications and Alchemist, and we ran some experiments to better understand the behaviour of these transfer times with respect to message buffer sizes, matrix layouts, and network variability.

References

[1] A. Gittens, K. Rothauge, S. Wang, M. W. Mahoney, J. Kottalam, Prabhat, L. Gerhardt, M. Ringenburg, and K. Maschhoff, "Alchemist: An Apache Spark ¡=¿ MPI interface," in *Concurrency and Control: Practice and Experience: Special Issue of the Cray User Group (CUG 2018)*, 2018.

[2] A. Gittens, K. Rothauge, M. W. Mahoney, S. Wang, J. Kottalam, Prabhat, L. Gerhardt, M. Ringenburg, and K. Maschhoff, "Accelerating Large-Scale Data Analysis by offloading to High-Performance Computing Libraries using Alchemist," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '18)*, 2018, pp. 293–301.

[3] *Alchemist: An HPC Interface for Data Analytics Frameworks*, 2018. [Online]. Available: github.com/project-alchemist/

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: a unified engine for big data processing," in *Communications of the ACM*, vol. 59, 2016, pp. 56–65.

[5] J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Transactions on Mathematical Software*, vol. 39, pp. 1–24, 2013.

[6] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: dask.org

[7] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124 – 1139, 2011.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[9] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2018.

[10] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Application," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[11] Elemental Development Team, *Elemental DistMatrix documentation*, 2018. [Online]. Available: libelemental.org/documentation/dev/core/dist_matrix/DM.html

[12] NERSC Cori. [Online]. Available: nersc.gov/users/computational-systems/cori/