2020 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)

Accelerating GPU-based Machine Learning in Python using MPI Library: A Case Study with MVAPICH2-GDR

S. Mahdieh Ghazimirsaeed †, Quentin Anthony†, Aamir Shafi,
Hari Subramoni, and Dhabaleswar K. (DK) Panda
The Ohio State University
{ghazimirsaeed.3, anthony.301, shafi.16, subramoni.1, panda.2}@osu.edu

***Abstract*—The growth of big data applications during the last decade has led to a surge in the deployment and popularity of machine learning (ML) libraries. On the other hand, the high performance offered by GPUs makes them well suited for ML problems. To take advantage of GPU performance for ML, NVIDIA has recently developed the cuML library. cuML is the GPU counterpart of Scikit-learn, and provides similar Pythonic interfaces to Scikit-learn while hiding the complexities of writing GPU compute kernels directly using CUDA. To support execution of ML workloads on Multi-Node Multi-GPU (MNMG) systems, the cuML library exploits the NVIDIA Collective Communications Library (NCCL) as a backend for collective communications between processes. On the other hand, MPI is a de facto standard for communication in HPC systems. Among various MPI libraries, MVAPICH2-GDR is the pioneer in optimizing GPU communication.**

This paper explores various aspects and challenges of providing MPI-based communication support for GPU-accelerated cuML applications. More specifically, it proposes a Python API to take advantage of MPI-based communications for cuML applications. It also gives an in-depth analysis, characterization, and benchmarking of the cuML algorithms such as K-Means, Nearest Neighbors, Random Forest, and tSVD. Moreover, it provides a comprehensive performance evaluation and profiling study for MPI-based versus NCCL-based communication for these algorithms. The evaluation results show that the proposed MPI-based communication approach achieves up to 1.6x, 1.25x, 1.25x, and 1.36x speedup for K-Means, Nearest Neighbors, Linear Regression, and tSVD, respectively on up to 32 GPUs.

***Index Terms*—Machine Learning, cuML, MPI, GPUs**

I. Introduction

The last decade has witnessed unprecedented growth in data generated from diverse sources. These sources range from scientific experiments—like the Large Hadron Collider [1] and Sloan Digital Sky Survey [2]—to social media platforms [3] to personalized medicine [4]. A daunting challenge—common to these disparate Big Data applications—is to process vasts amount of data in a timely manner to gain vital insights and drive decision making. To aid with this quest for better understanding, there has been a resurgence of Machine Learning (ML) libraries, tools, and techniques that allow processing and extracting useful information from this data through iterative processing.

†These authors contributed equally to this work.

Some of the popular ML libraries employed for Data Analytics include Scikit-learn [5] and Apache Spark's MLlib [6]. These libraries are natively designed to support the execution of ML algorithms on CPUs. On the other hand, GPUs have emerged as a popular platform for optimizing parallel workloads because of their high throughput. This also makes them a good match for ML applications, which require high arithmetic intensity [7].

To take advantage of the performance offered by GPUs, NVIDIA has recently launched the RAPIDS AI [8] framework, which is a collection of open-source software libraries and APIs. The main goal of this effort is to enable end-to-end data science analytic pipelines entirely on GPUs. One of the main components of RAPIDS—within its data science ecosystem—is the cuML[9] library. This GPU-accelerated ML library is the GPU-counterpart of Scikit-learn and provides similar Pythonic interfaces while hiding the complexities of writing compute kernels for GPUs directly using CUDA. One of the main benefits of cuML is that it supports the execution of ML workloads on Multi-Node Multi-GPUs (MNMG) systems. For this, it takes advantage of a task-based framework called Dask [10], which allows the distributed execution of Python data science applications. Dask is comprised of a central scheduler, workers (one per GPU), and a client process.

When a cuML application is running in a MNMG configuration, workers need to communicate with each other. These communications are required at two stages: 1) the training data is distributed to all workers to execute the training stage (the fit() function), 2) the output of the training stage i.e. the model parameters are shared with all workers involved in the inference stage (the predict() function). These communications can either be point-to-point or collective. cuML takes advantage of Dask for handling the point-to-point communications while using NVIDIA Collective Communications Library (NCCL) to take care of collective communications.

Message Passing Interface (MPI) is considered the *de facto standard* for writing parallel applications on clusters. The MPI standard [9] provides efficient support for point-to-point and collective communication routines. The MVAPICH2-GDR[10] library is a pioneer MPI library that supports communication between GPU devices. MVAPICH2-GDR has been used to

978-1-6654-2291-8/20/$31.00 ©2020 IEEE
DOI 10.1109/MLHPCAI4S51975.2020.00010

Table I: Comparison of different ML libraries

Libraries	GPU Support	MNMG Support	Python Interface	High Performance
Scikit-learn [5]	✗	✗	✓	✗
Apache Spark's MLlib [6]	✗	✓	✓	✗
Apache Mahout [7]	✓	✓	✗	✗
RAPIDS cuML [8]	✓	✓	✓	✗
MPI [9]	✓	✓	✗	✓
Our paper	✓	✓	✓	✓

accelerate and scale [11] Deep Learning (DL) frameworks including TensorFlow [12] on large clusters. In order to leverage years of research and development behind the MPI standard and its compliant libraries, this paper introduces support for MPI-based communication for the cuML library. This is done using the MVAPICH2-GDR library that has efficient collective communication designs — including `MPI_Reduce()`, `MPI_Bcast()`, and `MPI_AllReduce()` — on GPU memory.

A. Motivation and Challenges

Table I compares different libraries that can be used to run ML applications. Among these libraries, Apache Mahout [7], RAPIDS cuML , and MPI support execution on GPUs. Note that NVIDIA has recently created a RAPIDS Accelerator [13] for Spark 3.0 that enables the launch of Spark applications on GPUs. However, Spark does not natively support GPU acceleration. Apache Mahout, however, has poor visualization and it does not support Python interface. It also has poor performance compared to the other libraries. The respective strengths of cuML and MPI are complementary. While cuML provides a Python interface and hides the complexities of CUDA/C++ programming from the user, MPI is well-known for its high performance in parallel applications. Therefore, the question we ask is: ***How can we combine the ease-of-use provided by cuML for running ML applications with the high-performance provided by MPI?***

As mentioned earlier, cuML utilizes NCCL for handling collective communications. It has some APIs at the CUDA/C++ layer for MPI communications, but these APIs are not available at the Python layer and cannot be applied by end users developing Python code. At the same time, MPI libraries have many years of research and development behind them. Among these libraries, MVAPICH2-GDR, in particular, provides efficient designs of collective communications for GPUs. We use the OSU microbenchmarks suite [14] to compare the performance of the MVAPICH2-GDR library with NCCL for different collective operations in Figure 1. We run the experiments on the Comet cluster from XSEDE. A detailed description of this cluster is provided in Section VII-A. We run the experiments for Allreduce, Bcast, and Reduce for the 4 Bytes to 16 KBytes message range. These collectives are extensively used by cuML algorithms within this message range—this will be further discussed in Section VII. Figure 1 shows that MVAPICH2-GDR is performing significantly better than NCCL. This brings up another question: ***How can we replace NCCL-based collective communications in cuML with MPI-based communications to take advantage of efficient and GPU-aware collective communication designs in MVAPICH2-GDR?***

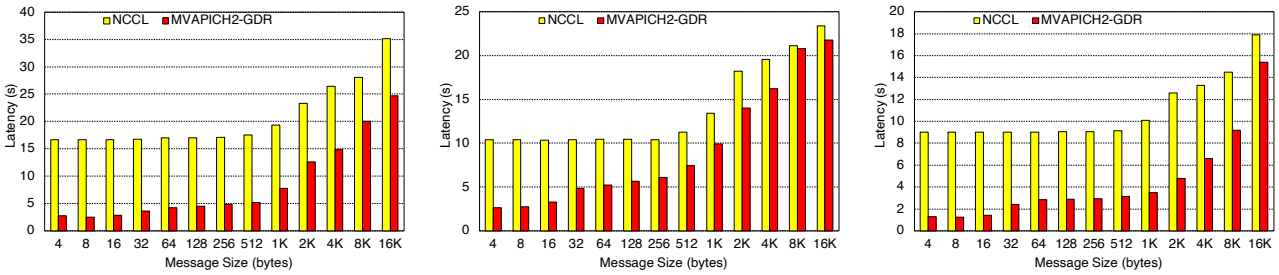
The cuML library supports training/inference based on several compute-bound ML algorithms. These include: 1) Clustering (like K-Means), 2) Dimensionality reduction (like Principal Component Analysis (PCA) and Truncated Singular Value Decomposition (tSVD)), 3) Ensemble methods (like Random Forests), and 4) Linear models (like Linear Regression). Data scientists should attempt to understand cuML in order to achieve the best possible performance on these algorithms. However, cuML—being a relatively new ML library—has not been studied well by the community. With this background in mind, the question is ***How can we provide performance characterization for GPU-accelerated cuML Algorithms and provide guidelines for data scientists to best take advantage of them?***

Each of the cuML algorithms mentioned above has a unique format for data and model features. Moreover, each cuML model has a unique set of hyperparameters that must be tuned for accuracy at every scale. In order to run cuML applications in a fast (time-to-solution) and accurate manner, a synthetic benchmarking suite as well as a hyperparameter tuning framework is required. This brings up another question: ***How can we provide a synthetic benchmarking suite for cuML algorithms in order to understand their scaling behavior?***

B. Contributions

The key contributions of this paper are as follows:

- We provide MPI-based communication support for GPU-accelerated cuML applications in Python. More specifically, we propose a Python API to take advantage of MPI-based communications for cuML applications.
- We provide in-depth analysis and characterization of the state-of-the-art cuML applications and identify regions that can be enhanced using MPI-based communications.
- We provide synthetic benchmarking of cuML algorithms to characterize their throughput behavior. Further, we apply Dask_ML's hyperparameter optimization library on the Higgs Boson dataset to ensure cuML's distributed algorithms maintain accuracy at scale.
- We provide a comprehensive performance evaluation and profiling study comparing MPI-based versus NCCL-based communication for cuML applications. The evaluation results show that with adding support for MPI-based communications, we gain up to 38%, 20%, 20%, and 26% performance improvement for K-Means, Nearest



(a) Allreduce (b) Reduce (c) Bcast

Figure 1: The performance of collective operations with MPI (MVAPICH2-GDR) vs. NCCL with 8 GPUs across 2 nodes on Comet cluster and for various collective operations: Allreduce, Reduce, and Bcast. MVAPICH2-GDR performs 84%, 74%, and 85% better that NCCL for Allreduce, Reduce, and Bcast, respectively.

Neighbors, Linear Regression, and tSVD, respectively on up to 32 GPUs.

The rest of the paper is organized as follows. Section II provides background information on different libraries that are currently used to run cuML applications (such as Dask and NCCL) and MPI. Section III discusses different ML applications that are supported by cuML and are targeted in this paper. This includes K-Means, Random Forest, Linear Regression, and truncated SVD (tSVD). Section IV provides detail on the synthetic benchmarking and hyperparameter optimization of these applications. In Section V, we provide a comprehensive overview of cuML's software stack and discuss the use of Dask and NCCL for handling different communications paths in cuML. The proposed MPI-based communication support is discussed in Section VI. Section VII characterizes state-of-the-art cuML applications and compares the performance of the proposed MPI-based approach vs the default NCCL-based communication design. We discuss related works in Section VIII. Finally, we conclude the paper in Section IX.

II. Background

In this section, we provide background information on the Dask, MPI, and NCCL libraries. These libraries are used to discuss and analyze the cuML software stack in upcoming sections.

A. Dask

Dask is an open source library for parallel computing in Python. It enables the Python ecosystem to scale into multi-core machines and distributed clusters. The main advantage of Dask is that it integrates well with analytic tools (such as Pandas, Scikit-Learn, Numpy, etc.) and provides ways to scale them on distributed clusters with minimal rewriting. In the past, Dask only supported execution on the host processors (CPUs). However, it has recently been integrated with the RAPIDS framework for processing cuDF data structures on GPU devices and also GPU-accelerated machine learning with cuML. For this, it distributes data and computation over multiple GPUs. These GPUs can be located on the same system or they can be distributed over a multi-node cluster.

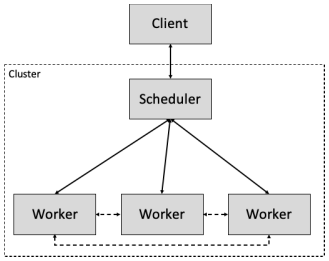


Figure 2: Dask Architecture

Figure 2 shows the Dask architecture. As can be seen in the figure, Dask consists of a client, a scheduler, and multiple worker processes. These processes communicate with each other to execute the Python application in a distributed manner. The scheduler is responsible for coordinating data access and sending tasks to workers for execution. It also manages the state of the workers. The workers are responsible for actual execution of the tasks. The scheduler along with N workers is called a Dask cluster. The Dask cluster in Figure 2 has 3 workers. As the figure shows, the workers are directly connected to each other for sending/receiving the data during the execution of parallel jobs. In the past, Dask only supported TCP for the communication between client, scheduler, and the workers. However, recently, it has been extended to take advantage of the UCX [15] library for communication between the processes.

The end-users submit their Python applications to the Dask cluster through the client process. The client process is connected to the scheduler and provides the execution code and the input data to the scheduler. Then, the scheduler divides up the input data and store the chunks on workers to achieve data parallelism.

B. MPI

MPI has been the de facto standard for communication in HPC systems, and is by far the dominant parallel programming model in large-scale parallel applications. In MPI, the concepts of groups and communicators are used to define the scope and context of the communications. A group is an ordered

set of processes with ranks 0 to $n-1$, where n is the number of ranks. When a program initializes, the rank number is assigned to each process. Communicators use groups to represent the processes that participate in each communication. Each process can be a member of one or more communicators.

MPI supports different types of communications including point-to-point and collective communications. The point-to-point communication is a basic mechanism in which only the sender and receiver take part in the communication. In collective communication, which is extensively used by the applications in cuML , messages are exchanged among a group of processes. Collective communications give this opportunity to the process to have one-to-many and many-to-many communications in a convenient, portable, and optimized way. Bcast, Reduce, and Allreduce are some examples of collective communications.

There are various MPI libraries available in the HPC community. Among them, MVAPICH2-GDR is specifically designed to improve the communication performance on GPU devices and provides efficient performance for GPU-enabled HPC and deep learning applications [16]. It has support for GPUDirect-RDMA, OpenPOWER with NVLink interconnect, CUDA-aware managed memory, and MPI-3 one-sided communication, among many other features.

C. NCCL

NCCL implements a subset of the collective communication routines. Specifically, NCCL provides optimized collective communication algorithms for NVIDIA GPUs. The available primitives for collective communication in NCCL are: Allgather, Allreduce, Reduce, Reduce-scatter, and Bcast. It is important to note that NCCL does not follow the MPI standard, and lacks support for many common MPI operations such as point-to-point, Gather, Scatter, and Alltoall. NCCL was created to meet the need for communication routines in common distributed deep learning workloads. Given the growth of Deep Learning applications and the volume of NVIDIA processors on HPC systems, NCCL has become a competitor to MPI for some applications.

III. Distributed Machine Learning Algorithms

This section discusses various ML algorithms that are currently available in cuML. This includes clustering algorithms (e.g. K-Means), dimensionality reduction (e.g. PCA and tSVD), ensemble methods (e.g. Random Forests), and linear models (e.g. Linear Regression) It also explains how these algorithms are parallelized in cuML under MNMG configuration.

Each algorithm in cuML is split into `fit()` and `predict()` functions that loosely take the place of training and inference functions. Namely, `fit()` acts on the training data (e.g. a dense matrix for K-Means) as input, and adjusts the weights of the initial model. After `fit()` has ended and the model is trained, `predict()` acts on the test data (e.g. a mapping from points to clusters for K-Means) as input, and provides a mapping to the output for each test sample.

A. K-Means

K-Means is an algorithm that separates n data points into k clusters by minimizing each cluster's $\sum ofsquarederror(SSE)$:

$$\sum_{n} \underset{\mu_j \in C}{min}\{||x_i - \mu_j||^2\}$$

where μ_j is the mean of the samples in cluster C. The extent of parallelism in Scikit-learn is in splitting the data into chunks, and processing each chunk with a separate thread. In cuML, however, the single-GPU K-Means model is copied onto each node, and the initial centroid is communicated to each worker GPU via a Bcast operation, and the total cluster cost and centroid selections are performed with one-time calls to Allreduce and Allgather, respectively. At each training iteration, the centroids from each Dask worker GPU are communicated with an Allreduce operation. Since the data communicated at each training step is small (the centroid information), the message size for each communication call is also small.

B. Random Forest

Random Forest is an ensemble model made up of decision tree classifiers. Decision tree classifiers predict a target variable's value by learning simple decision criteria from the training data's features. A decision tree may be easily explained as a set of yes or no questions posed to the input training variable. The decision tree makes a classification decision based on the answers to these questions. A random forest combines many individual decision trees, and takes a majority vote on the tree outputs to decide on the final classification.

In cuML, there is no collective communication used for Random Forest training at the time of writing this paper. Instead, Dask is used to partition the data over each worker GPU. Specifically, to train N trees with ω workers, each worker initializes N/ω trees and trains them on that worker's local data subset. cuML's distributed Random Forest algorithm takes an embarrassingly-parallel approach: all communication is carried out at the outset of training, and workers act independently once training has begun. Therefore, the communication overhead for random forests is small; we do not expect any performance difference between communication backends.

C. K Nearest Neighbors

Nearest-neighbors classification seeks to assign each data point based on a simple majority vote of its neighbors. An example of K Nearest Neighbors on a single worker with $K = 3$ is depicted below in Algorithm 1

In cuML's distributed implementation of K Nearest Neighbors, a subset of data is sent to each worker GPU with a call to Bcast, and the output of each local model is communicated at each global training step with a call to Reduce.

Algorithm 1 Example K Nearest Neighbors (with $K = 3$)

```
load(input_data)
K ← 3
/* Classify each point                                   */
foreach data_point_i in input_data do
    list ← {}  foreach data_point_j in input_data do
        /* Store distances                               */
        list.append(distance(data_point_i, data_point_j))
            /* Sort points by distance                   */
        list.sort()
    end
    /* Class set by closest k points                     */
    data_point_i.class ← majority_class(list[0 : k])
end
```

D. Linear Regression

Linear Regression is the classical problem of fitting a set of data points y with a linear combination of the predictors. In cuML, Linear Regression on multiple GPUs can take one of two forms: a standard data-parallel Linear Regression algorithm with two possible fit algorithms (SVD and Eig), or a model-parallel `Solver` class. For the standard data-parallel version, cuML provides the choice of fit algorithms: SVD and Eig. While computing the SVD (singular value decomposition) for a linear system of equations is stable, it tends to be much slower than finding the Eigen decomposition of the associated covariance matrix (the solution in cuML referred to as Eig). Nevertheless, for the purposes of this paper, we found SVD to be more useful for taking reproducible benchmarks. For more details on the SVD, refer to the next section on the tSVD algorithm in cuML. Linear Regression was parallelized in cuML by applying a Bcast operation at the outset of training and then applying a Reduce operation at each training step.

E. Truncated SVD (tSVD)

The Truncated Singular Value Decomposition (tSVD) is another widely-used method of dimensionality reduction (in addition to PCA) that is more suited to sparse matrices. Specifically, is a matrix factorization that generalizes the eigendecomposition of a matrix M, computing:

$$M \approx U_k \Sigma_k V_k^T$$

Where M is an $m \times n$ large matrix, U is an $m \times m$ unitary matrix, Σ is an $m \times n$ rectangular diagonal matrix, and V is an $n \times n$ unitary matrix.

tSVD is a less computationally intensive version of full SVD that only computes the k largest singular values of a large matrix M. Scikit-learn and cuML use a randomized matrix approximation algorithm originated by Halko et al. [17] that consists of two major steps: First compute an approximation to the range of M with randomization methods. That is, an intermediate matrix Q must be constructed with a small number of orthonormal columns such that:

$$M \approx QQ^T M$$

This approximate matrix Q may be computed by taking a collection of random vectors $\{\Omega_1 \Omega_2, ...\}$ and take the action of M on each vector. The resulting subspace of actions may be used to form another intermediate matrix Ω, which may be used to compute Q.

In the second step, a small intermediate matrix B is constructed such that $B = Q^T M$, and compute its SVD. Then, since $M \approx QQ^T M = Q(S\Sigma V^T)$, it is clear that $U = QS$ constitutes an approximation $M \approx U\Sigma V^T$. In cuML, tSVD is parallelized by performing distributed matrix computations. Specifically, tSVD was parallelized in cuML by applying a Bcast operation at the outset of training, and then applying a sequence of Reduce and Allgather operations to perform the matrix computation.

IV. Synthetic Benchmarking and Hyperparameter Optimization

In order to take distributed training and accuracy performance data, the existing benchmarking and hyperparameter optimization implementations needed to be expanded to support a distributed workload. We were able to refer to the single-GPU benchmarking suite in cuML as a model for distributed synthetic benchmarking.

For synthetic benchmarking, we seek to maximize throughput without exceeding the memory bound of the worker GPUs. Therefore, cuML's GPU-equivalent of the popular Scikit-learn function `make_blobs()` was used to generate random samples to feed into each clustering algorithm (K-Means and K Nearest Neighbors). The `make_blobs()` function generates a set of random isotropic gaussian blobs for clustering applications. To simulate a real dataset, each class is allocated one or more normally-distributed clusters of points. User-set parameters may be tuned to modify the center and standard deviation of each generated cluster of points. Given the flexibility of the `make_blobs()` function, we also applied dimensionality reduction algorithms (tSVD) to the clustered data for our throughput experiments. During the initial throughput study in Figure 10 we set these parameters in tandem with the hyperparameters to maximize GPU memory allocation while forcing enough model complexity to keep the GPU and interconnect under load for the duration of training. For each algorithm in Figure 10, these parameters and the total number of training iterations was kept fixed for each scale.

For the classification applications in cuML (Random Forests), we applied `make_classification()` to create a random n-class classification problem. The cuML function `make_classification()` initially generates normally-distributed clusters of points with $std = 1$ about vertices of a K-dimensional hypercube with sides of length M, where K, M are user-defined. An equal number of clusters is assigned to each class, while inserting both interdependence and random noise into the data. Features are then horizontally stacked and split into informative (independent) and redundant (linear combinations of informative) features.

For the regression applications in cuML (Linear Regression), we applied `make_regression()` to generate a set of

regression targets as a random linear combination of features with noise. The user may set the levels of sparsity and whether the informative features are uncorrelated or follow a low rank-fat tail singular profile (where a few features account for most of the variance). For the purposes of our study, the default uncorrelated informative features are sufficient. Examples of each of the three synthetic benchmarking methods are depicted in Listing 1.

```
from cuml import make_blobs,
    make_classification, make_regression


make_blobs(n_samples, n_features, centers,
    n_parts, cluster_std)

make_classification(n_samples, n_features,
    n_clusters_per_class, n_informative,
    random_state, n_classes)

make_regression(n_samples, n_features,
    random_state)
```

Listing 1: cuML example code for generating synthetic data for clustering, classification, and regression applications, respectively

In order to ensure that accuracy is not affected by the distributed algorithms in cuML, we trained K-Means and Random Forest algorithms on the Higgs Boson dataset [18]. We applied the hyperparameter optimization suite in Dask_ML to maximize the accuracy achievable at each scale. In particular, we used the adaptive cross-validation algorithm `HyperbandSearchCV`, which is specialized for specialized processors like GPUs, where the search space of hyperparameters is large. Specifically, Hyperband optimizers seek to spend the most time training high-performing sets of hyperparameters. To achieve this, Hyperband optimizers initialize many models, start to train all of them, and then drop poor-performing models before high-performing models have finished training. One can think of Hyperband optimizers as a randomized optimizer that drops poor parameter sets before training is finished, thus saving resources. The Dask_ML implementation `HyperbandSearchCV` follows an embarrassingly parallel approach by assigning each model with a unique parameter set to a worker GPU. Since there are many more parameter sets than worker GPUs in our experiments, we prioritize parameter sets based on the model's most recent loss score. This allows `HyperbandSearchCV` to be more aggressive in dropping poor-performing parameter sets. This is in accordance with the findings of [19].

Listing 2 shows an example HPO run on the cuML K-Means algorithm with `HyperbandSearchCV`. In this example, we pass K-Means parameter ranges for the number of clusters ($n\_clusters$), the oversampling factor ($sample\_factors$), and the stopping tolerance (tol) to `HyperbandSearchCV`, which performs at most max_iter training iterations on the best-performing parameter set. Finally, the aggressiveness in culling off the different estimators is passed to `HyperbandSearchCV` via the user-parameter $aggressiveness$.

```
from cuml.dask.cluster.kmeans import KMeans as
    cuKMeans

model_kmeans = cuKMeans(init="k-means||",
    n_clusters=5, random_state=123,
    oversampling_factor=3)

clusters = [i for i in range(1,100)]
sample_factors = [i for i in range(1,5)]

params = {'n_clusters' : clusters
          'oversampling_factor' : sample_factors
          'tol' : loguniform(1e-5, 1e-3)}

search = HyperbandSearchCV(model_kmeans,
    params, max_iter=100, aggressiveness=3,
    random_state=123)
```

Listing 2: cuML example code for performing hyperparameter optimization on KMeans

V. Overview of the Software Stacks

A. RAPIDS

NVIDIA has recently launched RAPIDS AI which is a collection of open source software libraries and APIs. It's used to run end-to-end data science analytics pipelines entirely on GPUs. For low-level compute optimizations, it relies on NVIDIA CUDA primitives and user-friendly Python interfaces to expose GPU parallelism and high-bandwidth memory speed.

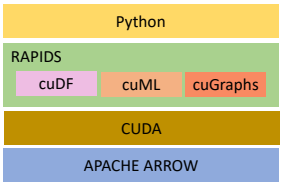
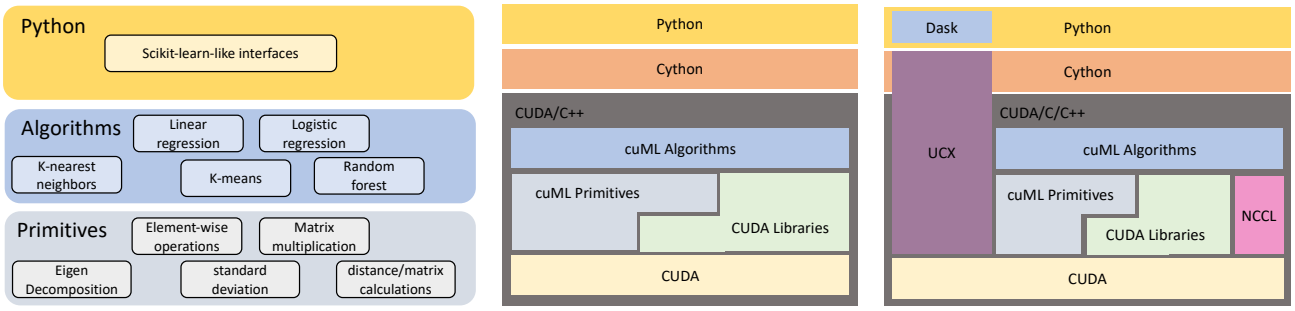


Figure 3: RAPIDS software stack

Figure 3 shows a high-level overview of the RAPIDS stack. As can be seen in this Figure, RAPIDS is built on top of CUDA and is under the standard specification of Apache Arrow [20]. Apache Arrow is a cross-language development platform for in-memory analytics. It defines a language-independent columnar memory format for flat and hierarchical data. It provides efficient analytic operations on both CPUs and GPUs.

RAPIDS has three main components: 1. cuDF which is a pandas-like dataframe manipulation library and is used for data preparation, 2. cuML which is a collection of machine learning libraries and provides GPU versions of algorithms available in scikit-learn, and 3. cuGraph which is an accelerated graph analytics library. The Python layer is built on top of these CUDA/C++ layers and data scientists can easily use it without worrying about the complexities of underneath layers. In this paper, we mainly focus on the cuML library as we aim to improve the performance of machine learning applications.



(a) cuML components (b) cuML software stack (c) cuML software stack in a distributed setting

Figure 4: cuML components and software stack

B. cuML Library

cuML is a suite of fast, GPU-accelerated machine learning algorithms within the RAPIDS data science ecosystem. cuML is designed for data science and analytical tasks. It enables users to run traditional ML tasks on GPUs without going into the details of CUDA programming. Figure 4(a) shows three main components of cuML: primitives, machine learning algorithms, and the Python layer. Primitives are reusable building blocks for building machine learning algorithms. They are common for different machine learning algorithms. Linear algebra is a big part of these primitives which includes element-wise operations, matrix multiplication, eigen decomposition, etc. The primitives also include statistical functions (such as the mean and standard deviation), random number generation, distance/matrix calculations, etc. These primitives are used in the second layer to build different machine learning algorithms such as Random Forest, K-Means, Linear Regression, etc. Finally, there is a Python layer that provides a Scikit-learn-like interface to expose these algorithms to data scientists.

The cuML library allows execution of ML workloads on a variety of platforms with three configurations: 1) a single GPU, 2) a system with multiple GPUs called Single-Node Multi-GPUs (SNMG), and 3) multiple systems with multiple GPUs called Multi-Node Multi-GPUs (MNMG). Figure 4(b) shows the software stack of cuML in a system with single GPU. The primitives and machine learning algorithms are built on top of CUDA libraries in the CUDA/C++ layer. Thrust, nvGraph, cuBlas, and cuRand are some example of these CUDA libraries. The CUDA/C++ layer is wrapped to the Cython layer to expose the cuML algorithms. The Python layer is used to integrate cuML with other tools in the community such as Numpy and cuDF and as mentioned earlier, it provides a Scikit-learn-like interfaces to the user.

C. cuML Library in a Distributed Setting

One of the main advantages of cuML is its support of distributed execution on MNMG systems. For distributed runs, cuML takes advantage of Dask. Dask enables the cuML code in Python to run in parallel across many nodes. To do this, Dask creates a task graph and divides up the code between workers. cuML also uses NCCL-based communication, especially for collective communications between the workers. This way, cuML takes advantage of both the simplicity and flexibility of Dask and the collective algorithms provided by NCCL.

Figure 4(c) shows cuML software stack in a distributed setting. As can be seen in the figure, both Dask and NCCL are used for communications in cuML . NCCL is mainly used for the collective communications between the workers in the `fit()` function. On the other hand, Dask is used for communication between the scheduler and workers while also handling point-to-point communications between workers. For example, Dask is used for sending the model parameters from one worker to the others in the `predict()` function. The reason Dask is used at this stage is that at the time of sending the model parameters, the root does not have any knowledge on how the data is going to be distributed among the workers/processes. More specifically, this communication is happening among a limited number of workers which are not known beforehand.

Another observation from this figure is that Dask uses UCX underneath to handle the communications. In other words, Dask uses UCX to pass CUDA objects between the workers. The advantage of UCX is that it provides the best communication performance to Dask based on the cluster's available hardware.

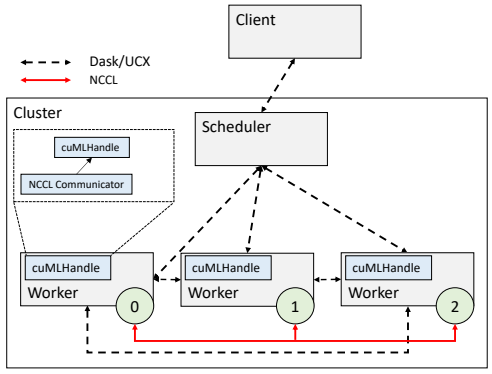


Figure 5: Dask and NCCL Communication paths in cuML

Figure 5 shows Dask and NCCL communication paths for cuML in a cluster with three workers. The black dotted

lines show Dask communication paths, while the red lines show NCCL communication paths. A NCCL communicator is created across the worker processes for taking advantage of collective communication support in NCCL. The NCCL communicator is then injected to `cumlHandle`. `cumlHandle` is a class in cuML that is used to manage resources needed for running machine learning algorithms. As can be seen in Figure 5, `cumlHandle` is used by all workers.

Once the NCCL communicator is attached to the `cumlHandle`, it can be used as the input of `fit()` and `predict()`. This way, the `fit()` and `predict()` functions take advantage of NCCL for running collective communications. Note that `fit()` has the most communication overhead in cuML algorithms. Figure 6 shows the procedure of injecting a NCCL communicator to `cumlHandle` in Python. As can be seen in the figure, this is done through `inject_nccl_comms_py()`. Once the NCCL communicator is attached to the `cumlHandle`, it is passed as the input of `fit()`.

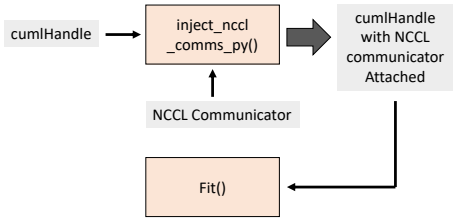


Figure 6: Injecting NCCL communicator to cumlHandle

VI. MPI-based communication support in cuML

In this section, we explain adding MPI-based communication support for cuML. More specifically, we discuss how to take advantage of an MPI library for running collective communications in the `fit()` function and replace the existing NCCL-based communications. Figure 7 shows the software stack of cuML with MPI-based communication support. We use MVAPICH2-GDR as the MPI library in our work due to its efficiency and high-performance on GPU clusters, but any other MPI library can be used. We also take advantage of mpi4py [21] as a Python binding library for MPI so that it can be used for running cuML applications written in Python.

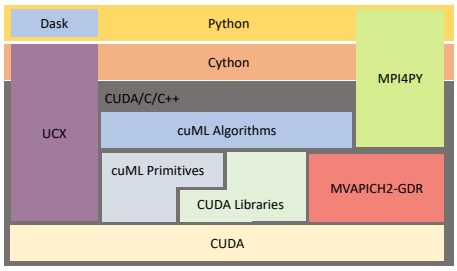


Figure 7: cuML software stack with MPI-based communication

As discussed in Section V-C, to support NCCL-based communications in cuML, a NCCL communicator should be injected to `cumlHandle`. A similar thing should be done for MPI communicator to add support for MPI-based communications. cuML has a function `initialize_mpi_comms` which attaches an MPI communicator to `cumlHandle`. This function is defined in the CUDA/C++ layer, so it cannot be used directly in the cuML applications written in Python. To be able to practically use this function for running cuML applications, we have developed a Cython wrapper `MPI_plugin.pyx` shown in Figure 8. This wrapper defines a new function `inject_mpi_comms_py()` which injects MPI communicator to `cumlHandle` in Python. It requires mpi4py to get an MPI communicator from `cumlHandle` and the function `initialize_mpi_comms`. For including `cumlHandle` and `initialize_mpi_comms`, cuML and CUDA libraries are linked to the wrapper through `setup.py`. We also linked MVAPICH2-GDR which is required by mpi4py. After compiling `MPI_plugin.pyx`, we have an extension module that injects MPI communicator to `cumlHandle` in Python. Listing 3 shows the details of defining `inject_mpi_comms_py()` in `MPI_plugin.pyx`.

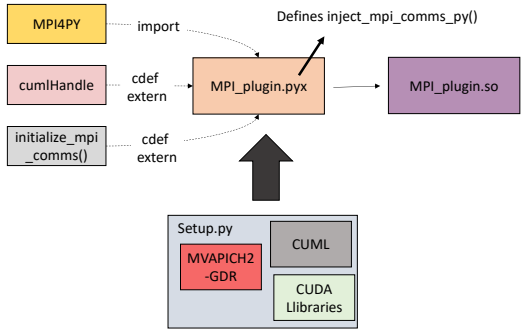


Figure 8: A Cython wrapper `MPI_plugin.pyx` for injecting an MPI communicator to `cumlHandle` in Python. `MPI_plugin.pyx` defines a new function `inject_mpi_comms_py` that wraps `initialize_mpi_comms` for Python. For this, `MPI_plugin.pyx` requires mpi4py to get an MPI communicator from `cumlHandle` and `initialize_mpi_comms`.

```
1  cimport mpi4py.MPI as MPI
2  cimport mpi4py.libmpi as libmpi
3
4  cdef extern from "cuml/cuml.hpp" namespace "ML":
5      cdef cppclass cumlHandle:
6          cumlHandle()
7
8  cdef extern from "cuML_comms.hpp" namespace
       "ML":
9      void initialize_mpi_comms(cumlHandle,
           mpi_comm)
10
11 def inject_mpi_comms_py(handle, MPI.Comm comm):
12     handle_ = <cumlHandle*> handle.getHandle()
13     initialize_mpi_comms(handle_, comm.ob_mpi)
```

Listing 3: `MPI_plugin.pyx`

Listing 4 shows how `inject_mpi_comms_py()` is used to inject an MPI communicator to `cumlHandle` in K-Means. Here, we show the modifications we made to K-Means as an example. We have made similar changes to other cuML algorithms to make them use MPI-based communications. The listing shows a part of `_func_fit()`. This function is executed by all the worker processes in the cluster. First, we import the module `MPI_plugin` that we have created from our Cython code in Listing 3. In `_func_fit()`, we create MPI communicator using mpi4py (Lines 3 and 4). This communicator is created between worker processes that are calling `_func_fit()`. Then, we import `Handle` from cuml.commom.handle (Line 6) and use it to create a `cumlHandle` object `handle` (Line 7). In Line 8, we call `inject_mpi_comms_py()` which we have defined in `MPI_plugin.pyx`. This function gets two inputs, the MPI communicator and the cuML `handle` and attaches the communicator to `handle`. Then, we pass the new `handle` to `cumlKMeans()` (Line 12). `cumlKMeans()` takes care of running the algorithm. It uses an MPI communicator that is attached to `handle` for executing the collective communications. Note that in Listing 4, we show the parts of `_func_fit()` that we have modified to be able to use the MPI-based communications. Other parts of `_func_fit()` have not been changed, so we did not include it in Listing 4.

```
1  import MPI_plugin
2  def _func_fit():
3      From mpi4py import MPI;
4      mpicomm = MPI.COMM_WORLD
5
6      from cuml.common.handle import Handle
7      handle = Handle()
8      MPI_plugin.inject_mpi_comms_py(handle,
           mpicomm)
9      .
10     .
11     .
12     cumlKMeans(handle, ...)
```

Listing 4: Injecting an MPI communicator to `cumlHandle` in `_func_fit()`, which is a part of the KMeans code

VII. Performance Characterization

A. Experimental Setup

We performed all experiments on the Comet cluster at the San Diego Supercomputer Center. The GPU partition on Comet is composed of 36 nodes each with 4 NVIDIA Pascal (P100) GPUs. Each P100 has 16 GB HBM2 memory and is connected to other nodes on the cluster via Infiniband FDR. Table II shows detailed specifications of the Comet cluster. We use cuML v0.15 compiled with CUDA 10.1, NCCL 2.7.8-1, and MVAPICH2-GDR 2.3.4.

B. Experimental Results

First, we compare the baseline performance of different ML algorithms when they are executed on a single CPU vs a single GPU. Figure 9 shows the results for K-Means, Random Forest, Nearest Neighbors, tSVD, and Linear Regression. Scikit-learn and cuML are used for the experiments on CPU and GPU, respectively. As can be seen in Figure 9, GPU performs significantly better than CPU for all the algorithms, as expected.

Table II: Hardware specification of the SDSC Comet cluster

Specification	SDSC Comet
Number of Nodes	36
Processor Family	Xeon Haswell
Processor Model	E5-2680 v3
Clock Speed	2.5 GHz
Sockets	2
Cores Per socket	12
RAM (DDR4)	128 GB
GPU Family	NVIDIA Pascal P100
GPUs	4
GPU Memory	16 GB (HBM2)
Interconnect	IB-EDR (56G)

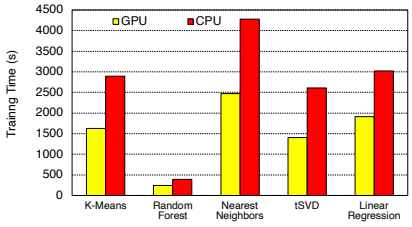
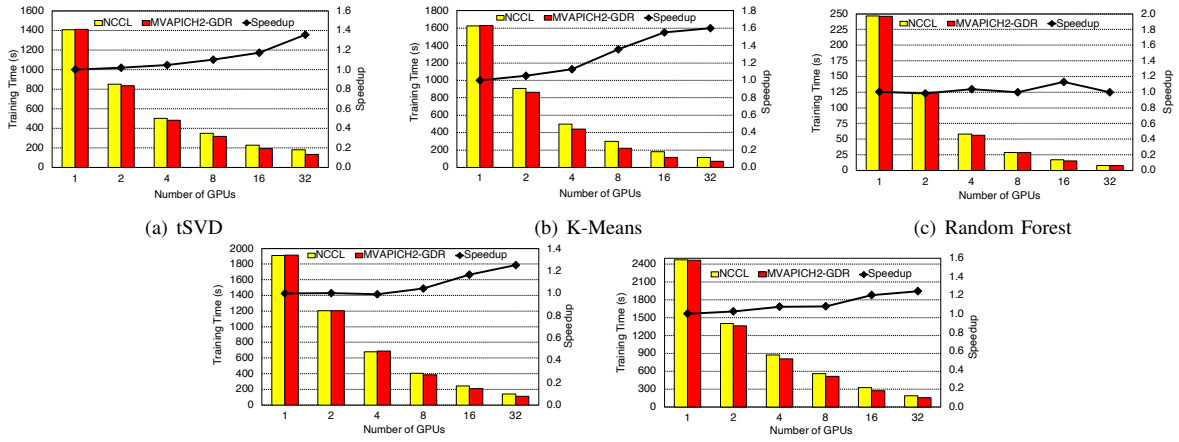


Figure 9: Training Time of different ML algorithms with GPU vs CPU on Comet cluster. For all the algorithms, the training time with GPU is significantly better than CPU.

Now that we have confirmed the efficiency of running ML algorithms on a GPU, we discuss the performance of ML algorithms on GPUs under a distributed or MNMG setting. Figure 10 shows the results on up to 32 GPUs across 8 nodes on the Comet cluster. We compare the performance of NCCL vs MPI-based communication with MVAPICH2-GDR. One observation from Figure 10 is that as we increase the number of GPUs, the training time reduces. This shows the scalability of cuML in the distributed setting. Another observation from the figure is that MVAPICH2-GDR is performing better than NCCL for almost all algorithms, and the performance improvement increases as we increase the number of GPUs. This is because the communication requirement of each algorithm is increased as the scale increases. The improved training performance shows that the MPI-based approach provides better scalability compared to the NCCL-based design. The only ML algorithm that performs the same with NCCL and MVAPICH2-GDR is Random Forest. The reason for this is that cuML does not use any collective communication for training Random Forest. Therefore, there is no room to take advantage of MVAPICH2-GDR's efficient design for this algorithm. In order to better understand why MVAPICH2-GDR performs better than NCCL at all scales, we collected the message size of each NCCL collective call in a training run for both K-Means and Nearest Neighbors on 2 Comet nodes (8 GPUs). From this profiling



(a) tSVD (b) K-Means (c) Random Forest

(d) Linear Regression (e) Nearest Neighbors

Figure 10: Training Time for GPU-Accelerated Distributed Algorithms using cuML.

information in Figure 11, we can see that the vast majority of collective calls are for the small to medium message size range. Note that we observed the same behavior for other cuML algorithms that use collectives such as tSVD and Linear Regression. We believe this to be the case for two primary reasons:

- For many distributed cuML algorithms like K-Means and Nearest Neighbors, the update information at each training step consists of many small messages (e.g. for K-Means, only the centroid information is shared between workers at each step). This allows the cuML algorithms to scale to many GPUs without significantly increasing the ratio of communication to computation
- At the Dask level, each message is chunked before distributing to individual workers. This increases the volume of collective calls while reducing the size of each individual message

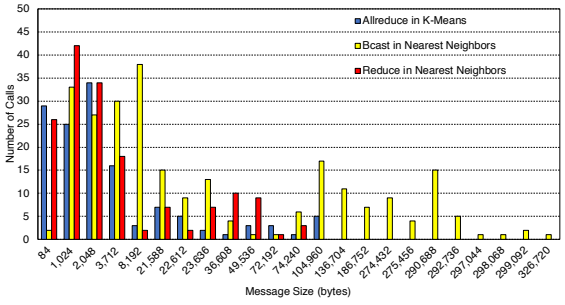
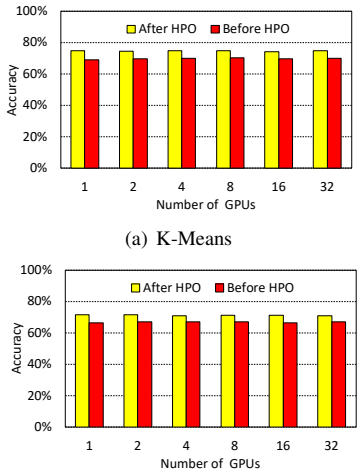


Figure 11: Profile of different collective operations in cuML algorithms: K-Means and Nearest Neighbors.

Given that the communication overhead for cuML algorithms consists of many small messages, we can take advantage of MVAPICH2-GDR's improved small-message algorithms compared to NCCL, as in Figure 1. Further, while cuML's small-message strategy does keep communication overheads low for increasing node counts, the increased number of collective calls allows MVAPICH2-GDR to improve its speedup compared to NCCL for a large number of worker GPUs, as depicted in Figure 10. Finally, we applied the hyperparameter optimization (HPO) library discussed earlier in section IV to the real-world Higgs dataset [18]. We used 30% of the dataset to perform HPO at each scale (1-32 GPUs). The accuracy on the test dataset was taken with the default model parameters for Random Forest and K-Means, then HPO was performed embarrassingly parallel via Dask_ML's `HyperbandSearchCV` across the job's worker GPUs. The accuracy results are depicted in Figure 12. The accuracy with the optimal parameters for K-Means and Random Forests show an approximate 5% improvement over the default model. We used K-Means and Random Forests on Higgs because:



(a) K-Means

(b) Random Forest

Figure 12: Accuracy Numbers

- They are representative of different communications in cuML (K-Means following a data-parallel approach, and Random Forests being embarrassingly parallel)

- Both algorithms have enough hyperparameters to stress the HPO implementation in Dask_ML, while having a broad parameter space to enable accuracy benefits

VIII. Related Work

González et al. [22] present a review of bagging and boosting ML algorithms. XGBoost [23][24] is a gradient boosting library that supports both distributed—through Dask—and GPU-based execution. H203 [25] is another ML library that is capable of distributed computation. H204GPU [26] is a variant with shared memory GPU support. While the computational support for ML workloads has been around for a long time in the form of software library and packages, the support for distributed execution on a cluster of GPUs is still in its infancy. As we noted above, XGBoost is one such library that provides boosting algorithms. However, a wide range of new and existing ML algorithms are still being investigated for efficient Multi-Node Multi-GPU execution—this is the main motivation for the development of the cuML library. Deep Learning frameworks like TensorFlow [12] and PyTorch [27] has support for some ML tools and techniques. However, we only focus on specialized ML libraries in this paper. Raschka et al. [28] provide a survey of machine learning in Python, including Scikit-learn training on CPUs and the RAPIDS ecosystem. They also discuss the need and motivation for the cuML library. The RAPIDS framework—and the cuML library—has been gaining traction in the community as a viable option to support the execution of performance-hungry applications on a cluster of GPUs. For example, Napoli et al. [29] applied RAPIDS [30] and Dask [31] ecosystems to analyze data for geophysics simulations. While all of these studies exploit the RAPIDS framework with cuML to exploit multiple NVIDIA GPUs, none of them provide insight into improving communication performance, nor explore other viable communication options. We fill this gap—in this paper—by proposing MVAPICH2-GDR as an alternative to NCCL in the Multi-Node Multi-GPU setting. We also attempt to characterize our evaluation for these GPU-aware messaging libraries to gain insights into scaling behavior of ML applications in the cuML library.

IX. Conclusion and Future Work

In this paper, we add support for MPI-based communications for cuML applications in Python. For this, we propose a Python API that takes advantage of MPI calls to handle collective communications between processes in cuML . Moreover, we provide a synthetic benchmarking suite and in-depth analysis of cuML applications to understand their behavior and identify the regions that can be improved using MPI-based communications. We believe these analysis and characterizations provide a comprehensive guideline for cuML application developers and data scientists to take the most advantage of the features provided by cuML. Finally, we compare the performance of the proposed MPI-based communication approach with NCCL-based communication design that is used by default in cuML. The experimental results on state-of-the-art cuML algorithms show that the proposed MPI-based communication approach gives up to 1.6x, 1.25x, 1.25x, and 1.36x speedup for K-Means, Nearest Neighbors, Linear Regression, and tSVD, respectively. For future work, we would like to explore the impact of the MPI-based communication support for real ML applications and larger data sets.

Acknowledgement

We would like to thank Arpan Jain for assisting us in the initial setup of cuML. This research is supported in part by NSF grants #1450440, #1565414, #1664137, #1818253, #1854828, #1931537, #2007991, and XRAC grant #NCR-130002.

References

[1] D. Guest, K. Cranmer, and D. Whiteson, "Deep learning and its application to lhc physics," *Annual Review of Nuclear and Particle Science*, vol. 68, pp. 161–181, 2018.

[2] M. R. Blanton, M. A. Bershady, B. Abolfathi, F. D. Albareti, C. Allende Prieto, A. Almeida, J. Alonso-García, F. Anders, S. F. Anderson, B. Andrews, and et al., "Sloan Digital Sky Survey IV: Mapping the Milky Way, Nearby Galaxies, and the Distant Universe," , vol. 154, p. 28, Jul. 2017.

[3] H. Ledford, "How facebook, twitter and other data troves are revolutionizing social science," *Nature*, vol. 582, pp. 328–330, 06 2020.

[4] H. Fröhlich, R. Balling, N. Beerenwinkel, O. Kohlbacher, S. Kumar, T. Lengauer, H. M. Maathuis, Y. Moreau, A. S. Murphy, M. T. Przytycka, M. Rebhan, H. Röst, A. Schuppert, M. Schwab, R. Spang, D. Stekhoven, J. Sun, A. Weber, D. Ziemek, and B. Zupan, "From hype to reality: data science enabling personalized medicine," *BMC medicine*, 2018.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, p. 1235–1241, Jan. 2016.

[7] R. Anil, S. Owen, T. Dunning, and E. Friedman, *Mahout in Action*, Manning Publications Co. Sound View Ct. 3B Greenwich, CT 06830, 2010. [Online]. Available: http://manning.com/owen/

[8] S. Raschka, J. T. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology

trends in data science, machine learning, and artificial intelligence," *ArXiv*, vol. abs/2002.04803, 2020.

[9] "MPI-3 Standard Document," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[10] MVAPICH2-GDR Development Team, "MVAPICH2-GDR User Guide," http://mvapich.cse.ohio-state.edu/userguide/gdr.

[11] A. A. Awan, A. Jain, Q. Anthony, H. Subramoni, and D. K. Panda, "HyPar-Flow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training using TensorFlow," *arXiv preprint arXiv:1911.05146*, 2019. [Online]. Available: http://arxiv.org/abs/1911.05146

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[13] NVIDIA, "RAPIDS Accelerator For Apache Spark," https://github.com/NVIDIA/spark-rapids.

[14] D. Panda et al., "OSU microbenchmarks v5.6.3," http://mvapich.cse.ohio-state.edu/benchmarks/.

[15] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[16] A. A. Awan, J. Bedorf, C.-H. Chu, H. Subramoni, and D. K. Panda, "Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation," *arXiv preprint arXiv:1810.11112*, 2018.

[17] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions," 2009.

[18] N. Becker, A. Dattagupta, E. Fajardo, P. Gali, B. Rhodes, B. Richardson, and B. Suryadevara, "Streamlined and accelerated cyber analyst workflows with clx and rapids," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 2011–2015.

[19] Scott Sievert, Tom Augspurger, and Matthew Rocklin, "Better and faster hyperparameter optimization with Dask," in *Proceedings of the 18th Python in Science Conference*, Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, Eds., 2019, pp. 118 – 125.

[20] Apache Software Foundation, "Arrow: Across-language development platform for in-memory data," https://arrow.apache.org.

[21] 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, new Computational Methods and Software Tools. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0309170811000777

[22] S. González, S. García, J. Del Ser, L. Rokach, and F. Herrera, "A practical tutorial on bagging and boosting based ensembles for machine learning: Algorithms, software tools, performance study, practical perspectives and opportunities," *Information Fusion*, vol. 64, pp. 205 – 237, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1566253520303195

[23] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[24] XGBoost Development Team, "XGBoost Library," https://xgboost.readthedocs.io/en/latest/.

[25] H2O Development Team, "H2O3: Distributed, fast, and scalable machine learning software," http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html.

[26] N. Gill, E. LeDell, Y. Tang, "H2O4GPU: Machine Learning with GPUs in R and Python," https://github.com/h2oai/h2o4gpu.

[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[28] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, p. 193, Apr 2020. [Online]. Available: http://dx.doi.org/10.3390/info11040193

[29] O. O. Napoli, V. M. do Rosario, J. P. Navarro, P. M. C. e Silva, and E. Borin, "Accelerating multi-attribute unsupervised seismic facies analysis with rapids," 2020.

[30] NVIDIA, "RAPIDS: Open GPU Data Science Framework," http://rapids.ai.

[31] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.