SHMEM-ML: Leveraging OpenSHMEM and Apache Arrow for Scalable, Composable Machine Learning

Max Grossman[1](✉), Steve Poole[2], Howard Pritchard[2], and Vivek Sarkar[1]

[1] Georgia Institute of Technology, Atlanta, GA, USA
max.grossman@gatech.edu
[2] Los Alamos National Laboratory, Los Alamos, NM, USA

Abstract. SHMEM-ML is a domain specific library for distributed array computations and machine learning model training & inference. Like other projects at the intersection of machine learning and HPC (e.g. dask, Arkouda, Legate Numpy), SHMEM-ML aims to leverage the performance of the HPC software stack to accelerate machine learning workflows. However, it differs in a number of ways.

First, SHMEM-ML targets the full machine learning workflow, not just model training. It supports a general purpose nd-array abstraction commonly used in Python machine learning applications, and efficiently distributes transformation and manipulation of this ndarray across the full system.

Second, SHMEM-ML uses OpenSHMEM as its underlying communication layer, enabling high performance networking across hundreds or thousands of distributed processes. While most past work in high performance machine learning has leveraged HPC message passing communication models as a way to efficiently exchange model gradient updates, SHMEM-ML's focus on the full machine learning lifecycle means that a more flexible and adaptable communication model is needed to support both fine and coarse grain communication.

Third, SHMEM-ML works to interoperate with the broader Python machine learning software ecosystem. While some frameworks aim to rebuild that ecosystem from scratch on top of the HPC software stack, SHMEM-ML is built on top of Apache Arrow, an in-memory standard for data formatting and data exchange between libraries. This enables SHMEM-ML to share data with other libraries without creating copies of data.

This paper describes the design, implementation, and evaluation of SHMEM-ML – demonstrating a general purpose system for data transformation and manipulation while achieving up to a 38× speedup in distributed training performance relative to the industry standard Horovod framework without a regression in model metrics.

Keywords: OpenSHMEM · ML · Machine · Learning · Scalable · Composable · Python · Ecosystem · Data · Science

S. Poole et al. (Eds.): OpenSHMEM 2021, LNCS 13159, pp. 111–125, 2022.
https://doi.org/10.1007/978-3-031-04888-3_7

1 Motivation

Data science and machine learning techniques have found broad applications, from proxy modeling in scientific applications to consumer recommendation engines to autonomous vehicles.

Most DS/ML frameworks are written to maximize programmability and portability, sacrificing performance. For example, most are written for Python, an extremely flexible but also interpreted programming language with high overheads. Pandas, a popular Python library for data scientists, mostly follows a copy-on-write semantic for mutating large n-dimensional arrays. This can lead to massive memory consumption on moderately-sized datasets. While this trade-off makes sense for small-scale projects, this causes problems for even simple data processing, exploration, and visualization workflows on the large-scale datasets that are common place today.

As a result, several efforts have explored taking well-known techniques and frameworks from the HPC community and applying them to DS/ML frameworks to yield both productive and high performance domain specific libraries/languages. These past works generally fall in to two buckets: (1) efforts to transparently use HPC frameworks underneath existing, industry-standard DS/ML frameworks, or (2) effort to replace existing DS/ML frameworks with new ones built with HPC technologies from the start.

1.1 Related Work: Using HPC Frameworks Under Existing DS/ML Frameworks

For example, in [11] the authors used OpenSHMEM [7] to accelerate distributed Caffe training jobs of the LeNet Solver network by replacing the existing MPI-based gradient exchange with equivalent OpenSHMEM operations. While this yielded a 30% improvement in training time over the existing implementation, this application of HPC technologies ignores the rest of the data science workflow. Projects like this one focus on a relatively small segment of the data science workflow (in this case, model gradient updates). Additionally, given that these optimizations are generally done at the lowest level of the data science software stack, they may miss optimizations that are only possible when higher level semantics are exposed.

1.2 Related Work: Novel HPC DS/ML Frameworks

On the other hand, there are several recent projects that aim to offer an all new data science software stack built on top of HPC technologies.

Legate Numpy [2] aims to offer a numpy-like [6] interface for multi-dimensional array processing on top of a high performance, distributed programming model called Legion [3]. Legate exposes a SPMD interface in Python, and a bridge to arrays stored in Legion (called logical regions) that allows for Python programmers to interact with Legion arrays in a similar manner to how they would interact with a Numpy array. While the programming model will be

familiar, the authors call out that only a "a subset of the full NumPy API" is currently supported. To our knowledge, there would also be no straightforward way to use Legate arrays with other Python libraries (e.g. Tensorflow, scikit-learn).

Arkouda [5] is another example of a high performance data science framework, in this case built on top of Chapel [4]. It aims to offer "distributed arrays with parallel primitives", a "familiar interactive interface", and "smooth integration with mature HPC code". In the case of Arkouda, the high level architecture is a Chapel-based cluster communicating with a Jupyter/Python client. In this way, the user can run their analyses in an easy-to-use and familiar Python environment most of the time but still ship larger kernels to a massive, distributed environment when needed (while accepting that the functionality supported in that larger environment is also more limited).

In both the case of Arkouda and Legate, we can see some challenges with these approaches. While Legate tries to offer a familiar programming model and Arkouda supports single-threaded Python execution on the client, both approaches essentially ignore the existing and massive DS/ML Python ecosystem of libraries and tools that users may expect to have access to (even in an HPC, distributed environment).

1.3 Contributions

SHMEM-ML is a new distributed nd-array and distributed inference/training machine learning system built on top of OpenSHMEM, exposing productive C++ & Python APIs, and leveraging Apache Arrow to support integration with the broader Python ecosystem. SHMEM-ML is available open source at https://github.com/agrippa/shmem_ml.

The remainder of this paper is structured as follows. Section 2 will cover the high level programming model and APIs of SHMEM-ML, as well as walk through some simple examples of SHMEM-ML's usage. Section 3 will describe its implementation in detail. Section 4 will walk through some illustrative performance benchmarks, and Sect. 5 will wrap up with some discussion and conclusions.

2 Programming Model

2.1 Distributed SHMEM-ML Arrays

SHMEM-ML exposes C++ and Python APIs for:

1. Distributed nd-array creation, manipulation, and destruction.
2. Distributed training and inference of machine learning models, applied to SHMEM-ML's distributed nd-array abstractions.

Creating and mutating distributed SHMEM-ML arrays can be done concisely in both C++ and Python. Table 1 includes a few example SHMEM-ML APIs.

The full SHMEM-ML C++ and Python APIs is too long to be included inline, but in general SHMEM-ML arrays support:

Table 1. Example SHMEM-ML routines for creating, accessing, and manipulating SHMEM-ML arrays in C++ and Python.

Operation	C++	Python
Create a 1D array of length N initialized to zero	`ShmemML1D<float> arr(N, 0.0);`	`PyShmemML1DD(N)`
Create a 2D array of size MxN initialized to zero	`ShmemML2D<float> arr(M, N, 0.0);`	`PyShmemML2DD(M, N)`
Apply a function to each element of an array	`arr.apply_ip([] ... );`	`arr.apply(lambda ...)`
Access a single local or remote element	`arr.get(i);`	`arr.get(i)`

- Allocation of distributed one- and two-dimensional arrays of primitive types and arbitrary size, up to the limits of the machine being used.
- Applying custom functions element-wise.
- Getting or setting elements.
- Clearing arrays to a specified value.
- Atomically updating local or remote array elements.
- Global reductions across the entire array (e.g. sum reduction).
- Saving and restoring of arrays to disk.

2.2 SHMEM-ML Arrays with Third Party Python Libraries

Additionally, today SHMEM-ML arrays integrate with commonly used Python data science libraries, including numpy, scikit-learn [8], and keras [12]. Section 3 includes more details on the implementation of this integration. For example, you can use a numpy random number generation API to populate data in a distributed SHMEM-ML array:

```
from PyShmemML import rand
vec = rand(vec)
```

Under the covers, the above code snippet uses the `numpy.random.rand` interface to implement random number generation.

It is also possible to train and apply scikit-learn models on SHMEM-ML distributed arrays. In the example below, `Xtrain`, `Ytrain`, `Xvalid`, and `predictions` are all distributed SHMEM-ML arrays.

```
from PyShmemML import SGDRegressor
clf = SGDRegressor(max_iter=niters)

clf.fit(Xtrain, Ytrain)
predictions = clf.predict(Xvalid)
```

The same can be done with Keras:

```
from tensorflow import keras
from PyShmemML import Sequential

clf = Sequential()
clf.add(tensorflow.keras.Input(shape=(5,)))
clf.add(tensorflow.keras.layers.Dense(3, activation='relu'))
clf.add(tensorflow.keras.layers.Dense(1, activation='relu'))
opt = keras.optimizers.SGD(learning_rate=0.01)

clf.compile(optimizer=opt, loss='mse')

clf.fit(Xtrain, Ytrain, epochs=niters)
predictions = clf.predict(Xvalid)
```

The structure of the code above will be very familiar to any existing Python data scientists. However, behind the scenes the data and workload is being distributed across an OpenSHMEM-based cluster. At the same time, we are leveraging all of the existing software in the Python data science ecosystem by relying on third party libraries like scikit-learn and keras for algorithms like forward propagation, backward propagation, gradient calculation, optimizers, etc.

2.3 Client-Server vs. SPMD

One of the main differences between how data scientists and HPC programmers interact with high performance clusters today is in the fundamental parallelism model exposed to them. Most data scientists are familiar with a client-server style model, in which a single Python notebook or shell distributes work to a large cluster. This is also the approach taken in Arkouda. However, most HPC programmers are more familiar with SPMD-style programming as it generally offers better scalability by removing the bottleneck of distributing work from a single client. This is the approach taken by Legate.

While it is safe to assume that SPMD-style programming will be more scalable for most use cases, it is also important to meet data scientists where they are comfortable. As a result, SHMEM-ML supports both a client-server style interface and an SPMD-style interface.

By default, SHMEM-ML in Python runs in SPMD mode with each process executing the same Python program in parallel. Processes have access to a `PyShmemML.pe()` function to fetch their unique OpenSHMEM PE ID, and `PyShmemML.npes()` to fetch the number of running OpenSHMEM PEs.

To run in client-server mode, rather than launching the Python program using the `python` interpreter (e.g. `python foo.py`), the programmer uses a SHMEM-ML wrapper called `shmem_ml_client_server` (e.g. shmem_ml_client_server foo.py). Then, the SHMEM-ML program will be run with a single process distributing work to the entire cluster.

In this way, users in both the data science and HPC communities can choose the programming abstractions they are most comfortable with. In the case of Arkouda and Legate, each programming system dictated whether the programmer worked in client-server or SPMD mode.

3 Implementation

SHMEM-ML is built on top of a number of open source or third party software packages. This section offers a brief overview of the fundamental building blocks of SHMEM-ML, as well as how they are put together to support distributed arrays and integration with the broader Python ecosystem.

At a high level, SHMEM-ML uses:

- Apache Arrow [1] for in-memory data storage and zero-copy data exchange with third party Python libraries.
- OpenSHMEM [7] for distributed job creation, inter-process communication, and inter-process coordination.
- Tensorflow, Keras, scikit-learn, numpy and other Python data science libraries for the implementation of more algorithmically complex data science functionality such as training and inference of deep neural networks.

3.1 Background: OpenSHMEM

The OpenSHMEM library provides a single program, multiple data (SPMD) execution model in which N instances of the program are executed in parallel. Each instance is referred to as a processing element (PE) and is identified by its integer ID in the range from 0 to $N-1$. PEs exchange information through one-sided *get* (read) and *put* (write) operations that access remotely accessible *symmetric objects*. Symmetric objects are objects that are present at all PEs and they are referenced using the local address to the given object. By default, all objects within the data segment of the application are exposed as symmetric; additional symmetric objects are allocated through OpenSHMEM API routines. OpenSHMEM's communication model is unordered by default. Point-to-point ordering is established through *fence* operations, remote completion is established through *quiet* operations, and global ordering is established through *barrier* operations.

3.2 Background: Apache Arrow

Apache Arrow is an open community effort to define a universal in-memory data format for n-dimensional arrays. Arrow's aim is to enable zero-copy, efficient data exchange between different libraries regardless of language and without each library having to provide explicit support for every other library. Apache Arrow defines a number of commonly used objects and functionalities, including one-dimensional arrays, two-dimensional tables, and file I/O.

3.3 Background: Scikit-Learn, Tensorflow, and Horovod

scikit-learn and Tensorflow/Keras are industry standard libraries for training and applying data-driven or machine learned models. Scikit-learn focuses on providing classes for more classical and statistically-derived machine learning models

(e.g. linear regressors, support vector machines, random forests, gaussian mixtures). Tensorflow/Keras focus on more recent developments in deep learning models, making it simple and straightforward to create deeply layered models with a variety of built-in layer types supported (e.g. Dense, Convolutional, Pooling, Recurrent, Normalization). Custom layer types can also be added by programmers. While Keras was started as an independent framework for training deep learning models, it was eventually merged into Tensorflow in 2017 as an alternative API.

While scikit-learn does not support distributed training today, Tensorflow/Keras offer a number of options. The most commonly used framework for distributed training of Keras models is Horovod [9] which uses an efficient ring-allreduce method to distributed gradient updates while sitting on top of high performance communication libraries (e.g. MPI) when supported. The introduction of Horovod to the Tensorflow/Keras communities drastically improved the scalability and productivity of distributed training.

3.4 ND-Array Implementation

Today, SHMEM-ML distributed arrays are limited to being either one- or two-dimensional – in the future, this restriction could be lifted. In either case, the core data backing a SHMEM-ML array is an Apache Arrow data structure allocated on the OpenSHMEM symmetric heap. In the case of a one-dimensional array, we use Arrow's `FixedSizeBinaryArray` class which allows us to allocate a contiguous array of elements, each containing `sizeof(T)` bytes. In the case of two-dimensional SHMEM-ML arrays, we use Arrow's `Table` class to store columns of Arrow `Arrays`.

To have the backing allocations for Arrow's `Table` and `Array` classes allocated in the OpenSHMEM symmetric heap, we have also implemented a custom Arrow `MemoryPool` that supports `Allocate`, `Reallocate`, and `Free` functions that operate on memory regions in the symmetric heap. This custom memory pool is passed to the Arrow runtime when constructing a new `Array` or `Table`, and in turn the Arrow runtime calls it when memory is needed.

One-dimensional SHMEM-ML arrays are distributed in chunks across the available OpenSHMEM PEs, and two-dimensional arrays are chunked across rows. Today, the type of distribution and chunk size is chosen for the programmer – future work could extend this to support different data distributions (e.g. cyclic). SHMEM-ML also supports what we call "replicated" arrays. When allocated with a size `N`, they allocate `N` elements on every PE (rather than distributing them across PEs). In general, replicated arrays are useful when a programmer wants to update the local copy and then perform some type of a global sync of every PE's local updates (e.g. a global sum of all local values).

SHMEM-ML arrays include functions for looking up basic information on a distributed array, including the number of elements in the array, which PE stores a given element based on its index, and the starting/ending indices of elements stored on the local PE.

SHMEM-ML arrays also support a number of getter and setter APIs. For example, one-dimensional arrays support both getting and setting elements in the array using either global indices into the entire distributed array or local indices into the local chunk of the array. Some example one-dimensional APIs are included below. Under the covers, all remote operations are performed using OpenSHMEM APIs (e.g. `shmem_putmem` or `shmem_getmem`).

```
// A remote get based on the global index in the distributed array
inline T get(int64_t global_index);

// A remote set based on the global index in the distributed array
inline void set(int64_t global_index, T val);
```

The getter and setter methods above are not atomic (i.e. if two PEs try to set the same global index, the result is undefined). As a result, SHMEM-ML also supports atomic operations on elements of arrays. There are two implementations of atomic operations in SHMEM-ML: OpenSHMEM-based and message-based. OpenSHMEM-based atomic operations are directly implemented using OpenSHMEM atomics APIS (e.g. `shmem_longlong_atomic_fetch_add`. Message-based atomics are packaged up by the SHMEM-ML runtime as a small packet encoding the operation to be performed and sent in batches to the target PE through asynchronous mailboxes. In the cases of workloads performing large numbers of atomic operations, this approach increases latency of individual operations but can also drastically improve throughput. On the receiving side of an atomic message, updates are simply done using memory reads and writes – this means that the two types of atomics in SHMEM-ML are not atomic with respect to each other. Additionally, because message-based atomics are asynchronous an additional `sync` call is needed on the array in question to ensure all outstanding atomics have been sent and processed. Some example atomics APIs are included below.

```
// Perform an atomic compare-and-swap at the designated element.
// Return the previous value at that location.
T atomic_cas(int64_t global_index, T expected, T update_to);

// Perform an atomic compare-and-swap at the designated element,
// using the message-based atomics implementation.
void atomic_cas_msg(int64_t global_index, T expected, T update_to);

// Wait for all pending message-based atomics to complete on the
// target array.
void sync();
```

Finally, SHMEM-ML arrays also support global reductions performed on their contained elements (e.g. max reduction, sum reduction). In general, a local result is computed sequentially and then an OpenSHMEM reduction is performed to compute the global result based on each PE's local result. Some example reduction array APIs are shown below.

```
T max(T min_val);
T sum(T zero_val);
```

3.5 Client-Server Implementation

Section 2 described the difference between SPMD and client-server execution from the user's perspective. All that was needed to switch to a client-server architecture for SHMEM-ML was to use a special `shmem_ml_client_server` executable when launching your distributed Python program, rather than the standard `python` interpreter.

Under the covers, this custom executable does the following:

1. Initializes OpenSHMEM and the Python runtime (if we are running client-server mode from Python, and not from C++).
2. Sets a flag on each OpenSHMEM PE to indicate which are servers/workers, and which is the client. In general, we select PE 0 as the client.
3. Symmetrically allocates what we call a command mailbox on every PE. This is the only mechanism by which the client PE issues work to worker/server PEs. Every time a distributed operation occurs on the client PE (e.g. distributed array allocation, a distributed `apply`, a global reduction), coordination messages are sent from the client PE to all server PEs informing them of the distributed operation to be performed.
4. All PEs that are servers then enter a command loop, waiting on new incoming commands from the client and then performing the requested operations.
5. The client PE then launches the provided Python program using `PyRun_SimpleFileExFlags`. When it completes, it sends all servers a command to indicate that the program has completed and a collective `shmem_finalize` occurs.

Naturally, client-server mode faces some intrinsic scalability bottlenecks that SPMD mode does not. However, for programmers that are less familiar with an HPC-style programming environment it offers a more comfortable on-ramp to using SHMEM-ML.

3.6 Integration with Scikit-Learn and Tensorflow/Keras

SHMEM-ML's use of Apache Arrow enables zero-copy data exchange between SHMEM-ML and other Arrow-based libraries, including numpy, Pandas, scikit-learn, and Tensorflow. However, integrating into their workflows (particularly for model training) does require some added logic.

Supporting executing numpy functions on SHMEM-ML arrays is relatively straightforward. All SHMEM-ML arrays expose functions for (1) getting the local Arrow arrays backing them (`get_local_arrow_array`), and (2) updating their contents from another Arrow array (`update_from_arrow`). Arrow arrays can then be converted to or from numpy arrays, which can be passed to numpy's routines.

Model inference workloads are also relatively simple, and generally consist of applying a model element-wise to an input SHMEM-ML array after it has been converted to a numpy array via Apache Arrow. Below is a simplified example of the glue code between SHMEM-ML arrays and scikit-learn models for model inference.

```
def predict(self, x):
    # Convert the local chunk of our SHMEM-ML array to a Pandas
    # Dataframe using Apache Arrow
    x_arr = x.get_local_arrow_table().to_pandas(
            zero_copy_only=True, split_blocks=True)

    # Run the trained scikit-learn model on our local chunk,
    # producing a new numpy array as output
    pred = self.model.predict(x_arr)

    # Allocate a new distributed SHMEM-ML array to store the
    # result of the inference
    dist_pred = PyShmemML1DD(x.M())

    # Update the contents of dist_pred with the output of the model
    dist_pred.update_from_arrow(pyarrow.array(pred))

    # Return the new SHMEM-ML array containing the predictions
    return dist_pred
```

However, model training workloads require more extensive glue code between SHMEM-ML arrays and scikit-learn/Tensorflow models. In particular, because the models themselves are responsible for updating their weights but are not aware that they are being trained in parallel (i.e. that there are updates occurring on remote PEs to remote copies of the model), SHMEM-ML must (1) take over the iterative training process, (2) manage inter-process gradient exchange between iterations, and (3) rely on models' incremental training APIs to support iteration-by-iteration training updates. This is in contrast with how models are generally trained, by passing in the full training dataset and training for a large number of iterations.

However, this process is relatively uniform across frameworks. Indeed, the SHMEM-ML code base uses a single model training function to perform distributed training of both scikit-learn and Tensorflow models – with some model-specific logic plugged in (e.g. to fetch the weights from a given model type). A simplified version of that training function is shown below. Note that this implementation is likely making some assumptions about the type of model and type of optimizer being used for training, such that an averaging of weights on each iteration will yield convergence. While this approach has been tested for stochastic gradient descent optimizers, it may need customization for different types of model optimization.

```
def _training_driver(model, x_arr, y_arr, epochs, **custom_args):
    dist_weights_grad = PyReplicatedShmemML1DD(...)

    for it in range(epochs):

        # Rely on the model supporting incremental training
        model._fit_one_epoch(x_arr, y_arr, **custom_args)

        # Fetch the model's new weights as an arrow array, and
        # convert it to a distributed, replicated SHMEM-ML array
        arrow_weights = pyarrow.array(model._copy_weights())
        dist_weights_grad.update_from_arrow(arrow_weights)

        # Perform a sum reduction across all PEs on the model
        # weights following this iteration's updates
        dist_weights_grad.reduce_all_sum()

        # Extract a local numpy array containing the summed
        # weights, and normalize the new weights by number of
        # PEs (taking the mean of model weights across all PEs)
        all_weights_grads = dist_weights_grad \
                .get_local_arrow_array() \
                .to_numpy(zero_copy_only=True) / npes()

        # Update the model itself with the new average of all
        # model updates across all PEs
        model._update_weights(all_weights_grads)
```

4 Performance Evaluation

In this section, we will compare the performance and accuracy of models trained on scikit-learn, Tensorflow, and SHMEM-ML. All results were collected on the TACC Frontera machine's primary compute system [10]. Each node of Frontera includes a dual-socket Intel Xeon Platinum 8280 "Cascade Lake" CPU with 56 cores per node. Each node also includes 192 GB of DDR4 system memory. Nodes are connected by a Mellanox Infiniband HDR-100 interconnect.

SHMEM-ML was built using OSSS-UCX OpenSHMEM and Apache Arrow built from source code as of October 2020. GCC 9.1.0 was used. Tensorflow v2.1.0, scikit-learn 0.23.2, and Horovod 0.21.1 were used in this evaluation.

To evaluate SHMEM-ML's performance when training a scikit-learn model, we will compare the performance of training a scikit-learn linear regression model in a single-threaded Python process to distributed training in SHMEM-ML using both client-server and SPMD modes.

To evaluate SHMEM-ML's performance when training a Tensorflow model, we will compare between (1) Tensorflow single-node, (2) Tensorflow multi-node using Horovod, (3) SHMEM-ML SPMD, and (4) SHMEM-ML client-server.

4.1 Scikit-Learn

The source code for model training in both the SHMEM-ML and scikit-learn implementations of this benchmark are identical:

```
clf = SGDRegressor(max_iter=50)
clf.fit(X, Y);
predictions = clf.predict(X)
```

We use a synthetic dataset with 5 million samples and 5 32-bit floating point features per sample for the purposes of benchmarking.

Table 2 includes wall times for training on a single node for scikit-learn and SHMEM-ML. Note that running SHMEM-ML on a single node implies running one PE per core, and so the SHMEM-ML numbers are with 56-way parallelism. As expected, that added parallelization yields large speedups for SHMEM-ML in client-server and SPMD mode relative to scikit-learn (37.3× and 45.3×, respectively) with SPMD achieving slightly higher throughput. The sublinear speedup on a single node for even the SPMD version can be attributed to coordination and communication overheads required to exchange gradient updates between PEs on each iteration of training. This is commonly the largest challenge to training scalability, and a source of future work for SHMEM-ML (e.g. by leveraging more efficient communication patterns, similar to Horovod).

Table 2. Training performance running SHMEM-ML and scikit-learn on a single node

Framework	Training (s)	Training speedup
Single-threaded scikit-learn	509.83	1.0×
Client-server SHMEM-ML on one node	13.67	37.3×
SPMD SHMEM-ML on one node	11.26	45.3×

Figure 1 shows the execution time of SHMEM-ML's scaling while training the SGDRegressor model. We can observe the throughput benefits of SPMD mode, though both modes of execution fail to scale beyond 1,792 PEs (32 nodes).

4.2 Tensorflow

Like scikit-learn, the source code for the SHMEM-ML and Tensorflow implementations of this benchmark are identical:

```
clf = Sequential()
clf.add(tensorflow.keras.Input(shape=(nfeatures,)))
clf.add(tensorflow.keras.layers.Dense(1, activation='relu'))

opt = keras.optimizers.SGD(learning_rate=0.005)
clf.compile(optimizer=opt, loss='mse')
```

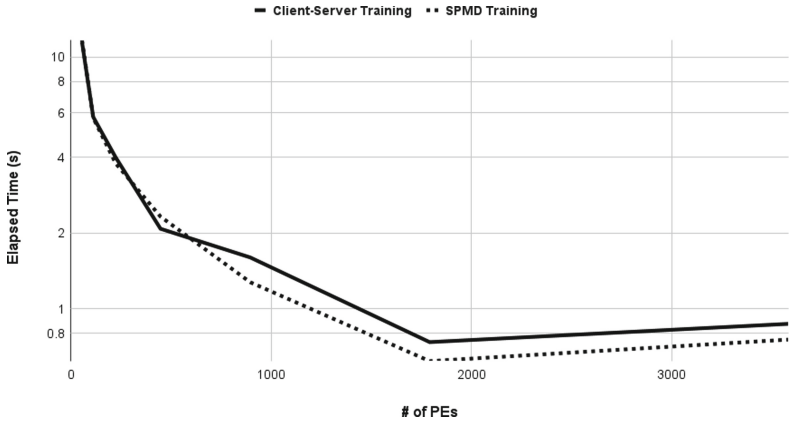

Fig. 1. SHMEM-ML execution time in client-server and SPMD modes for training. Note the log scale Y axis.

```
clf.fit(X, Y, epochs=niters, batch_size=128)
pred = clf.predict(X)
```

Like with scikit-learn, we use a synthetic dataset with 5 million samples and 5 32-bit floating point features per sample for the purposes of benchmarking.

There are additional hyperparameters in training of deep learning models which did not have to be considered in the scikit-learn comparison. In particular, we increase both the number of iterations/epochs in the distributed case and the batch size per iteration to yield better convergence. In general, the following tables and figures will report performance per iteration to ensure an apples-to-apples comparison. Table 3 describes the full set of hyperparameters tuned for our local and distributed jobs.

Table 3. Per framework hyperparameters

Framework	Iterations	Batch size
Single node tensorflow	40	32
Multi node tensorflow	5	128
Multi node SHMEM-ML	400	128

Table 4 includes wall times for training on a single node for Tensorflow and SHMEM-ML. Again, the parallelism added by SHMEM-ML yields large speedups in both client-server and SPMD mode relative to Tensorflow (111.98× and 116.57×, respectively). Numbers are not reported for Horovod+Tensorflow, as an OOM was encountered with only a single node.

Additionally, we ran experiments to compare the accuracy of the models produced by each framework – attempting to optimize hyperparameters for model

Table 4. Training performance running SHMEM-ML and tensorflow on a single node

Framework	Seconds per iteration	Speedup
Single node tensorflow	142.22	1.0×
Client-server SHMEM-ML on one node	1.27	111.98×
SPMD SHMEM-ML on one node	1.22	116.57×

metrics rather than for an apples-to-apples throughput comparison. Table 5 summarizes the results. Not surprisingly, SHMEM-ML's increased throughput enables more iterations and therefore better model metrics.

Table 5. Model metrics and performance

	Tensorflow	Horovod	SHMEM-ML
Iterations	40	130	500
Batch size	32	128	128
# Nodes	1	32	32
Total wall time	5899.15	2334.89	61.21
Seconds per iter.	147.48	17.96	0.12
RMSE	4.504500E−08	5.000000E−06	9.676908E−23

Finally, Fig. 2 shows the elapsed time per iteration of SHMEM-ML out to 32 nodes. While Horovod fails with an out of memory error below 16 nodes, its execution time per iteration at 16 and 32 nodes is much higher than SHMEM-ML. At 16 nodes, Horovod takes 65.65 s per iteration while SHMEM-ML SPMD takes 0.15 s per iteration. At 32 nodes, Horovod takes 14.13 s and SHMEM-ML SPMD takes 0.09 s.

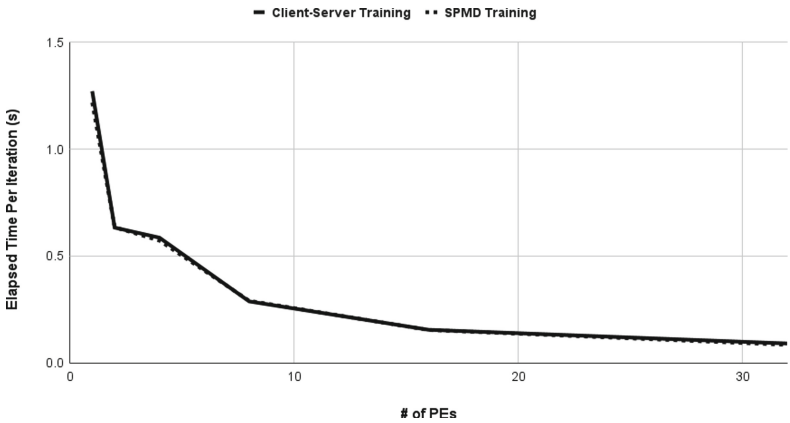

Fig. 2. SHMEM-ML execution time per iteration for training

5 Conclusions

SHMEM-ML leverages OpenSHMEM to accelerate data science and machine learning workflows. By focusing on a scalable distributed array data structure and composability with the existing Python data science ecosystem, SHMEM-ML aims to enable scalable end-to-end data science workflows – including data loading, data manipulation, model training, and model inference.

Acknowledgement. This research is part of the Frontera computing project at the Texas Advanced Computing Center. Frontera is made possible by National Science Foundation award OAC-1818253. This publication has been approved for public, unlimited distribution by Los Alamos National Laboratory, with document number LA-UR-21-29037.

References

1. Apache Arrow. https://arrow.apache.org/
2. Bauer, M., Garland, M.: Legate NumPY: accelerated and distributed array computing. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–23 (2019)
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2012)
4. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. App. **21**(3), 291–312 (2007)
5. Merrill, M., Reus, W., Neumann, T.: Arkouda: interactive data exploration backed by chapel. In: Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop, pp. 28–28 (2019)
6. NumPy. https://numpy.org/
7. OpenSHMEM application programming interface, version 1.5 (2020). http://www.openshmem.org
8. Scikit-Learn. https://scikit-learn.org/stable/
9. Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799 (2018)
10. TACC Frontera. https://www.tacc.utexas.edu/systems/frontera
11. Taylor, G., Ozog, D., Wasi-ur Rahman, M., Dinan, J.: Scalable machine learning with openshmem
12. Tensorflow Keras. https://www.tensorflow.org/api_docs/python/tf/keras