HeAT – a Distributed and GPU-accelerated Tensor Framework for Data Analytics

Markus Götz[§], Daniel Coquelin[†‡§], Charlotte Debus[*], Kai Krajsek[‡], Claudia Comito[‡], Philipp Knechtges[*], Björn Hagemeier[‡], Michael Tarnawa[‡], Simon Hanselmann[§], Martin Siggel[*], Achim Basermann[*] and Achim Streit[§]

[*]*Institute for Software Technology (SC)*
German Aerospace Center (DLR)
Cologne, Germany
{charlotte.debus, philipp.knechtges, martin.siggel, achim.basermann}@dlr.de

[†]*Institute of Bio- and Geosciences Agrosphere (IBG-3)*
Forschungszentrum Jülich (FZJ)
Jülich, Germany

[‡]*Jülich Supercomputing Centre (JSC)*
Forschungszentrum Jülich (FZJ)
Jülich, Germany
{k.krajsek, c.comito, b.hagemeier, m.tarnawa}@fz-juelich.de

[§]*Steinbuch Centre for Computing (SCC)*
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{markus.goetz, daniel.coquelin, simon.hanselmann, achim.streit}@kit.edu

***Abstract*—In order to cope with the exponential growth in available data, the efficiency of data analysis and machine learning libraries have recently received increased attention. Although corresponding array-based numerical kernels have been significantly improved, most are limited by the resources available on a single computational node. Consequently, kernels must exploit distributed resources, e.g., distributed memory architectures. To this end, we introduce HeAT, an array-based numerical programming framework for large-scale parallel processing with an easy-to-use NumPy-like API. HeAT utilizes PyTorch as a node-local eager execution engine and distributes the workload via MPI on arbitrarily large high-performance computing systems. It provides both low-level array-based computations, as well as assorted higher-level algorithms. With HeAT, it is possible for a NumPy user to take advantage of their available resources, significantly lowering the barrier to distributed data analysis. Compared with applications written in similar frameworks, HeAT achieves speedups of up to two orders of magnitude.**

***Index Terms*—HeAT, Tensor Framework, High-performance Computing, PyTorch, NumPy, Message Passing Interface, GPU, Data Analysis, Machine Learning, Dask, Neural Networks**

I. Introduction

The Python programming language has evolved into the de-facto standard for the data analytics and machine learning communities. Therein, the default choice for many frameworks is the SciPy stack [1], which is built upon the computational library NumPy [2]. NumPy offers efficient data structures and algorithms for vectorized matrix and tensor operations, allowing for the implementation of efficient numerical and scientific programs. More recently, deep-learning libraries such as TensorFlow [3] and PyTorch [4], both of which either implement or closely mimic the NumPy API, have begun to bridge the gap between pure CPU-based single-node processing and high-performance computing by offering GPU-accelerated kernels and simple Remote Procedure Call (RPC) style distributed computations.

These libraries work around Python's parallel computation limitations by implementing computationally intensive kernels in low-level programming languages, such as C/C++ or CUDA, and invoking the respective calls at run-time through some binding. This allows Python users to exploit powerful features like vectorization, threading, or the utilization of accelerator hardware. Still, most of these libraries are confined to the processing capabilities of a single computation node. Some libraries, such as Dask [5] or PyTorch's `dist` package, make it possible to write distributed programs based on RPCs. However, data decomposition, communication structure, workload balancing, etc., must be addressed by the user.

In response to these problems, we propose HeAT[1] – the Helmholtz Analytics Toolkit. HeAT is an open-source library offering a NumPy-like API for distributed and GPU-accelerated computing for general-purpose and high-performance computing (HPC) systems of arbitrary size. The central component of HeAT is the `DNDarray` data structure, an N-dimensional array transparently composed of computa-

This work is supported by the Helmholtz Association Initiative and Networking Fund (INF) under project number ZT-I-0003 and under the Helmholtz AI platform grant.

[1]https://github.com/helmholtz-analytics/heat

tional objects on one or more processes. The process-local objects are PyTorch Tensors, allowing HeAT functions to use both CPUs and GPUs. For distributed memory computing, communication between processes is crucial. To this end, HeAT provides a communications back end built on top of the Message Passing Interface (MPI) [6].

Due to the NumPy-like API of HeAT, existing NumPy programs can be quickly and easily converted into distributed HeAT applications. This design allows for users to adapt existing codes easily. Furthermore small-scale program prototypes can be developed, which can be transitioned transparently to an HPC systems without major code or algorithmic changes. Distributed HeAT applications are typically faster and their memory limitations are those of the whole system, rather than those of one node. As a result, HeAT facilitates the algorithmic development and efficient deployment of large scale data analytics and machine learning applications.

The remainder of this paper is organized as follows. Section II will present related work in the field of (distributed) array computation. Section III will explain HeAT's programming model, array, and communication design concepts. Here we will also highlight some of the unique features of HeAT that set it apart from other libraries. In Section IV, an empirical performance comparison with Dask, HeAT's primary competitor, is presented. Within this section, HeAT is shown to perform significantly better in all tested areas. Section V discusses the advantages and limitations of HeAT's programming model with respect to other frameworks. Finally, Section VI concludes the presented work and offers a glimpse into the future developments planned for HeAT.

II. Related Work

NumPy is arguably the single most important Python library for numerical calculations; scikit-learn [7] is its most widely adopted machine-learning counterpart. The typical NumPy/scikit-learn implementation involves a single CPU, with selected operations transparently using multiple cores. Running NumPy on multiple CPUs requires a specific configuration and usage of GPUs is not supported.

Modern machine learning and deep learning libraries (e.g. PyTorch [4], MXNet [8], and TensorFlow[3]) typically mimic NumPy's syntax to some degree and support distributed computations for certain parallelization models. For example, PyTorch allows explicit message passing with MPI, NCCL, and Gloo back ends, thus providing support for both multi-CPU and multi-GPU systems. Both PyTorch and TensorFlow allow remote computation via the RPC protocol and support AD in this mode. However, their approaches are primarily focused on performing embarrassingly parallel computations across multiple computing nodes. Packages such as Horovod [9] focus on offering a singular algorithm, i.e., data parallel model training, but do not target general distributed array-based computations.

Another approach to distributed computing uses lazy evaluation to increase the performance of tensor computations by operator fusion (e.g. Spartan [10], Bohrium [11], Grumpy [12], and Weld [13]). Instead of evaluating each line of code sequentially, the code is analyzed prior to execution to assess if operations can be fused together, then optimized code is generated before execution. The performance gained by operator fusion can quickly be dominated by the overhead required by task distribution. Finally, several widely used libraries, such as Legate [14], Ray [15], and Dask [5], adopt dynamic task scheduling for parallel execution of NumPy operations on CPU and GPU HPC systems.

TABLE I: Support for distributed memory computing and automatic differentiation within currently available Python libraries for machine learning.

Package	*Multi CPU*	*Single GPU*	*Multi GPU*	*NumPy API*	AD	Ref.
PyTorch	✓	✓	✓	✓	✓[a]	[4]
Legate	✓	✓	✓	✓		[14]
Dask	✓			✓		[5]
Intel DAAL	✓					[16]
TensorFlow	✓	✓	✓	✓	✓[a]	[3]
MXNet	✓	✓	✓	✓		[8]
DeepSpeed	✓	✓	✓	✓		[17]
DistArray	✓					[18]
Bohrium	✓	✓				[11]
Grumpy	✓		✓			[12]
JAX	✓	✓	✓	✓		[19]
Weld	✓		✓			[13]
NumPywren	✓					[20]
Arkouda	✓					[21]
GAiN	✓					[22]
Spartan	✓					[10]
Phylanx	✓					[23]
Ray	✓	✓	✓	✓		[15]
HeAT	✓	✓	✓	✓	✓	

[a]Based on RPC, no MPI support.

Table I summarizes the current possibilities for distributed memory high-performance machine learning within the Python landscape. In most cases support for parallel computation implies the availability of tools that make communication among processing units possible. Generally, transparent distributed memory computing is not supported in the aforementioned high-level APIs. HeAT has been developed to alleviate these shortcomings.

III. Design and Implementation

A. Programming Model

The Helmholtz Analytics Toolkit is an open-source library that implements a NumPy-like API for data structures, functions, and methods for array-based numerical data analytics and machine learning. An example can be seen in Listing 1. HeAT realizes a single-program-multiple-data (SPMD) programming model [24] using PyTorch and MPI. Additionally, the framework's processing model is inspired by the bulk-synchronous parallel (BSP) [25] model.

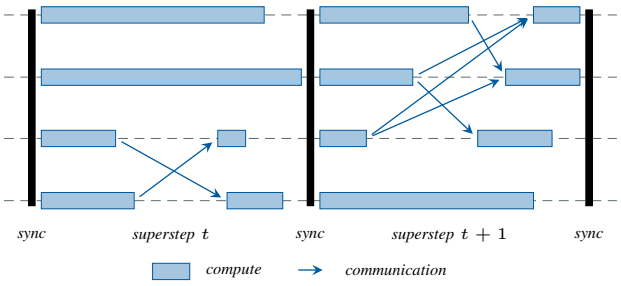


Fig. 1: The BSP-inspired parallel processing model utilized by HeAT. Computation steps are marked as light blue blocks, possible communication as blue arrows, implicit or explicit synchronization points as black vertical bars.

Listing 1: Implementation of a function calculating the standard deviation of an array, demonstrating the API compatibility between NumPy and HeAT.

```
1 import heat as ht
2
3 def standard_deviation(a, axis=0):
4     return ht.sqrt((a - a.mean(axis)) ** 2)
```

Framework computations proceed in a series of hierarchical supersteps, each consisting of a number of process-local computations and subsequent inter-process communications. In contrast to the classical BSP model, communicated data is available immediately, rather than after the next global synchronization. In HeAT, global synchronizations only occurs for collective MPI calls as well as at the program start and termination. A schematic overview is depicted in Fig. 1.

The process-local computations are implemented using PyTorch as the array-computation engine. Each computation is processed eagerly, i.e. when issued to the interpreter. The scheduling onto the hardware is controlled by the respective runtime environment of PyTorch. For the CPU back end, these are the synchronous schedulers of OpenMP [26] and Intel TBB [27]. For the GPU back end, it is the asynchronous scheduler of the NVidia CUDA [28] runtime system. HeAT provides the MPI "glue", utilizing the `mpi4py` [29] module, for the communication in each superstep. Users can freely access these implementation details, although it is neither necessary nor recommended to modify the communication routines.

B. DNDarrays

At the core of HeAT is the Distributed N-Dimensional Array, `DNDarray` (cf. Listing 2). The `DNDarray` object is a virtual overlay of the disjoint PyTorch tensors which store the numerical data on each MPI process. A `DNDarray`'s data may be redundantly allocated on each node or decomposed into equally sized chunks, with a maximum difference of one element along a single axis.

This data distribution strategy aims to balance the workload between all processes. During computations, API calls may arbitrarily redistribute data items. However, completed operations fully automatically restore the uniform data distribution.

To steer the data decomposition and other parallel processing behaviour, HeAT users can utilize a number of additional attributes and parameters:

- `split`: the singular axis, or dimension, along which a `DNDarray` is to be decomposed (see Fig. 2 and Listing 2) or `None`, if redundant copy;
- `device`: the computation device, i.e. CPU or GPU, on which the `DNDarray` is allocated;
- `comm`: the MPI communicator for distributed computation (Section III-C);
- `shape`: the dimensionality of the global data;
- `lshape`: the dimensionality of the process-local data

Listing 2: A `DNDarray` distributed across three processes as illustrated in Fig. 2(b).

```
 1 import heat as ht
 2 a = ht.zeros((5, 4, 3), split=0)
 3 a.shape
 4 [0/3] >>> (5, 4, 3)
 5 [1/3] >>> (5, 4, 3)
 6 [2/3] >>> (5, 4, 3)
 7 a.lshape
 8 [0/3] >>> (2, 4, 3)
 9 [1/3] >>> (2, 4, 3)
10 [2/3] >>> (1, 4, 3)
```

As stated, process-level operations on `DNDarrays` are performed via PyTorch functions, thus employing their C++ core library `libtorch` to achieve high efficiency, where available. Interoperability with external libraries such as NumPy and PyTorch is self-evident. Data contained in a NumPy `ndarray` or a PyTorch `Tensor` can be imported into a `DNDarray` via the `heat.array()` function with the optional `split` attribute. In the opposite direction, data exchange with NumPy is enabled by the `DNDarray.numpy()` method.

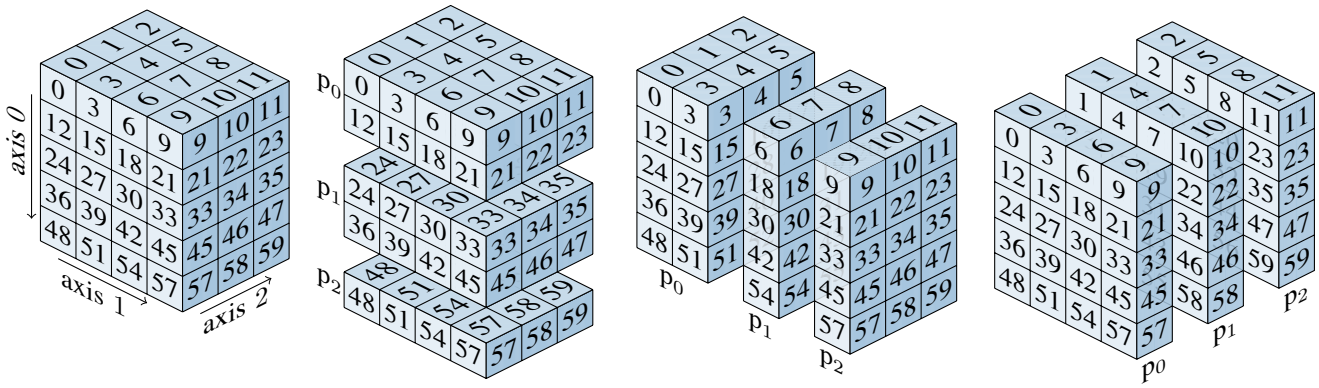
`DNDarray` can reside in a node's main memory for the CPU back end or, if available, in the VRAM of GPUs. Individual `DNDarrays` can be assigned to hardware devices via the `device` attribute or the default device can be defined as shown in Listing 3.

Listing 3: Programmatic ways of allocating `DNDarray` on different devices.

```
 1 import heat as ht
 2 # a single allocation
 3 a = ht.zeros((1,), device="gpu")
 4 a
 5 >>> tensor([0.], device="cuda:0")
 6
 7 # setting a default device
 8 ht.use_device("gpu")
 9 b = ht.ones((1,))
10 b
11 >>> tensor([1.], device="cuda:0")
```

C. Distributed Computation

Many algorithms using a distributed `DNDarray` will require communication. HeAT has a custom MPI-based communication layer composed of wrappers of point-to-point and global MPI functions. It utilizes the python library `mpi4py` [29], which offers an interface to most common MPI



(a) `split=None` (b) `split=0` (c) `split=1` (d) `split=2`

Fig. 2: Distribution of a 3-D `DNDarray` across three processes: (a), `DNDarray` is not distributed, i.e., `split=None`, each process has access to the full data; (b), (c), and (d): `DNDarray` is distributed along axis 0, 1 or 2 (`split=0`, `split=1`, or `split=2`, respectively). An example for case (b) is available in Listing 2. In each case, the data chunk labeled p_n resides on process n, with $n = 0$, 1, or 2.

implementations and enables the communication of contiguous Python buffer objects (e.g. NumPy arrays). In cases where CUDA-aware MPI is available, this also allows for communications to be performed directly between multiple GPUs. Without CUDA-aware MPI, data must be copied from the GPU to the CPU, sent to another CPU, then copied to the target GPU. This increases the communication overhead, as well as the run time, of some functions. The `DNDarray` memory representation is encoded in the one dimensional buffer via strides (steps between elements) along the respective dimension. A main challenge in communicating an arbitrarily split `DNDarray` is the preservation of this data structure. The HeAT communication module internally handles buffer preparation as the interface between the `DNDarray` and the `mpi4py` functionality.

For point-to-point communications (e.g. `send`, `receive`), buffer preparation is trivial as the data can be sent contiguously from one process and unpacked by the receiving process. More considerable efforts must be made for communication involving collective operations. For gathering operations (e.g. `gather`, `allgather`), the node-local `Tensor` sent by each process must have the correct memory layout, which is dependent on the split axis of the `DNDarray`. For scattering operations (e.g. `scatter`, `all-to-all`), the data chunks must be packed correctly along the split axis before distribution.

HeAT addresses the packing issues by creating custom MPI data types, which wrap the local `Tensor` buffer. First, the `DNDarray`'s dimensions are permuted such that the dimension along which data assembling or distribution should take place is the first dimension. Then, custom data types are created via the MPI function `Create_vector` to iteratively pack the dimensions from the last to the first. The individual data types at each dimension are defined via the `DNDarray`'s strides. The creation of such a buffer is schematically shown in Fig. 3. Here, a split `DNDarray` is assembled to `split=None` via the `allgather` function using custom send and receive buffer objects.

With this internal buffer handling, HeAT offers a unified interface that provides communication without exposing the internal data representation. Based on the MPI layer, a `resplit` function is provided to change the split axis of a `DNDarray` if required. Re-splitting a `DNDarray` adheres to load balancing, i.e., the data is uniformly distributed across processes as previously stated. However, caution must be taken when using `resplit` as it is based on global MPI communication functions, thus requiring both significant communication and local memory.

D. Unique Features

As a result of this design concept, HeAT offers a number of unique features, which distinguish it from existing libraries. Here, we briefly mention two of the more important.

1) Parallel Pseudo-Random Numbers: In many machine learning methods random guesses are used for initial values. To make this a reproducible process independent of the number of processes, HeAT implements a parallel pseudo-random number generator (pPRNG) building on counter-based PRNGs [30]. The core idea is to encrypt an easily and independently reproducible sequence, e.g. an ascending vector of natural numbers, with a symmetric encryption process where the random number seed is used as the encryption key. Through data decomposition of the sequence ($\mathcal{O}(1)$), the random number generation can be parallelized in a scalable manner and produces an identical sequence of random numbers independent of the processor count.

HeAT utilizes a eight-round 32-/64-bit Threefry [30] encryption process for generating uniformly distributed pseudo-random integer sequences. The corresponding floating point values can be obtained by masking out the sign and exponent bits and retaining the mantissa bits. Finally, normally distributed random numbers are usually derived by rejection

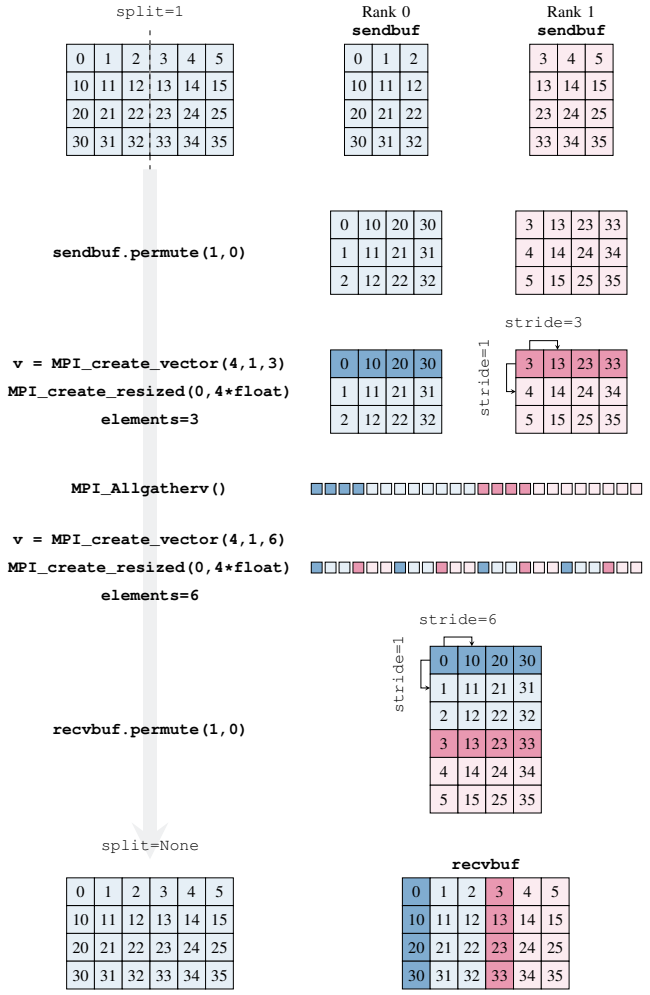


Fig. 3: Internal handling of a `resplit(None)` operation on a two-dimensional `DNDarray` with `split=1` in HeAT, i.e., data replication on all nodes. It depicts the on-the-fly creation of MPI datatypes for the strided access into the output and input buffers.

sampling methods like the Box-Muller transform [31]. In counter-based pPRNGs, this approach is either invalid due to redrawing already used counter numbers, thus biasing the random numbers, or it is computationally expensive due to global redrawing rounds. HeAT obviates this problem by implementing a direct conversion on top of generalized exponential distributions [32].

2) Distributed Automatic Differentiation: The most distinguishing feature of a machine learning library is that the majority of routines are automatically differentiable (AD), thus allowing for the use of gradient-based optimization algorithms during training. Since HeAT utilizes PyTorch's process-level commands, the process-level functions are mostly differentiable. However, the communication routines are not directly differentiable. In order to enable reverse-mode AD, i.e., backpropagation, HeAT uses differentiable wrappers for its communication routines that are transparent to PyTorch. These wrappers are embedded into the directed acyclic graph (DAG) that PyTorch builds for the backward step, enabling us to leverage the existing AD mechanics in PyTorch.

Let us consider a simple point-to-point communication example. A blocking send and receive operation can be interpreted as an assignment from a variable x at process p_x to another variable y residing on processes p_y. The send operation corresponds to a receive operation in the adjoint mode, and vice versa. It is possible to construct these pairs of operations for all MPI communication primitives, thus enabling fully automated differentiation. Using this mechanism one can construct not only data-, but also model-parallel and pipelined high-level differentiable algorithms.

IV. Performance Results

A systematic approach is utilized for the performance evaluation of HeAT. The results are compared with Dask [5], the library currently dominating the landscape of distributed computing in Python. Four algorithms are benchmarked: *k*-means in IV-C1, the forward propagation of a neural network in IV-C2, least absolute shrinkage and selection operator (LASSO) regression in IV-C3, and spectral clustering in IV-C4. Details about these algorithms can be found in their respective sections.

The benchmarking experiments are divided into weak scaling and strong scaling experiments. In the former, the workload with respect to the algorithmic complexity per process remains constant as the amount of computing resources increases. In the latter, the total workload remains constant as the amount of computing resources increases. Computational resources are expressed in the number of nodes.

Benchmarking results for weak scaling are presented as throughput per process, i.e., number of floating point operations per second, normalized by the maximum execution time of all processes, where the number of floating point operations can be either measured or approximated. If FLOPS cannot be measured, than results are presented as inverse run time in order to maintain trend behavior. For strong scaling, results are presented as the execution speedup compared to a NumPy-based implementation. This demonstrates the effectiveness of both HeAT and Dask against a baseline.

Each data point represents the average outcome of a total of nine runs, with the error bars indicating the empirical standard deviation. A warm-up run preceded the first run and the result is not included in the measurement. A fractional number of nodes refers to the usage of the equivalent fraction of a node's resources.

A. Execution Environment

The experiments were run on a machine learning HPC system comprised of 15 compute nodes at the Jülich Supercomputing Centre (JSC). Each node is equipped with two 12-core Intel Xeon Gold 6126 CPUs, 206 GB of DDR3 main memory and four NVIDIA Tesla V100 SXM2 GPUs with 32 GB VRAM per card. The GPUs communicate node-internally via an NVLink interconnect. The system is optimized for

TABLE II: Software packages used for performance benchmarks.

General		**Python**	
Package	***Version***	***Package***	***Version***
CUDA	10.2	dask	2.12.0
GCC	8.3.0	dask-ml	1.2.0
HDF5	1.10.5	dask-mpi	2.0.0
Intel Cluster Studio XE	2019.03	heat	0.3.0
PAPI	5.7	mpi4py	3.0.3
ParaStationMPI	5.2.2-1	numpy[a]	1.15.2
Python	3.6.8	python-papi	5.5.1.5
		sklearn	0.22.2
		torch	1.4.0

[a]using `Intel MKL 2019.1`.

GPUDirect communication across node boundaries, supported by 2x Mellanox 100 Gbit EDR InfiniBand links.

The software environment for benchmarking is summarized in Table II. At the time of writing, CUDA-aware MPI is not functioning on the system. Thus, all experiments were performed via MPI communication over host memory, resulting in extra copy operations for the multi-GPU measurements. We expect the adoption of CUDA-aware MPI communication to significantly reduce the overhead of communication operations. Additionally, PAPI [33] does not accurately measure the number of operations for Dask. Therefore, the operation count for Dask assumes that it corresponds with the number of operations required by an equivalent NumPy calculation. This slightly underestimates the total number of performed FLOPS as copy and communication operations will not be included. However, the difference is negligible as the computations required in each experiment are on the order of several TFlop.

The Dask execution model envisages one scheduler process and multiple worker instances. The scheduler sends workload via serialized RPC calls to the workers. The actual program code is provided in a separate script that connects the scheduler to the workers via a `dask.distributed.Client` instance. The discovery of the scheduler is done manually by passing an IP address or via information in files on a shared filesystem. Networking between the processes builds on network sockets and utilizes Infiniband using TCP over IB. Each worker maintains its execution state by writing into journaling directories.

B. Datasets

Three publicly available datasets were chosen to demonstrate the effectiveness of HeAT for different data characteristics and to mimic common use-cases. For the benchmarking of the *k*-means algorithms (Section IV-C1) and the forward propagation of a neural network (Section IV-C2), we utilized the Cityscapes dataset [34]. It contains 5 000 high-resolution images with fine-grained annotations. Each image is 2 048×1 024 pixels with three 256 bit RGB color channels per pixel, which have been flattened and combined into a short-fat matrix with 5 000×6 291 456 entries. For weak-scaling runs each process has 300 rows of the matrix. For strong scaling runs, the first 1 200 rows are used.

For the LASSO regression benchmark (Section IV-C3), parameters from the EURopean Air pollution Dispersion-Inverse Model(EURAD-IM) [35] have been used as input variables. The EURAD-IM is an Eulerian meso-scale chemistry transport model as part of the Copernicus Atmosphere Monitoring Service (CAMS)[2]. The regression targets of our experiment are the errors of ozone forecasts of the model at measurement sites[3], such that the LASSO regression infers the dependency of different model parameters on the forecast error. For the experiment, 10^7 data points and 100 parameters of the model have been chosen and stored in a tall-skinny matrix.

For the spectral clustering benchmark (Section IV-C4), the SUSY dataset was chosen [36] as a tall-skinny dataset. It contains 5 000 000 samples from Monte Carlo simulations of high-energy particle collisions. Each sample has 18 features, consisting of kinematic properties measured by the particle detectors and high-level functional derivations of those measurements [37]. The computational load of spectral clustering grows quadratically with the number of samples. For the weak scaling experiments sample numbers were thus increased by the square root of the number of processes involved. The first 25 820, 36 515, 51 640, 73 030, and 100 000 samples were used for N =1, 2, 4, 8 and 15 nodes, respectively. For experiments on one node with one or two GPUs, 12 910 and 18 258 samples were used. The strong scaling measurements were performed using the first 40 000 samples of the dataset.

All data sets were converted from their original sources into data matrices and stored as single-precision floats in an HDF5 file [38]. While Dask and HeAT utilize parallel I/O via `h5py` [39], they handle data decomposition differently. Dask offers an automatic data decomposition scheme and the manual specification of data chunk sizes. Dask has been tested with both automatic chunking (their recommended setup) and with a tuned chunking where the chunk size mirrors HeAT's data decomposition. All measurements with HeAT load the data with `split=0` (cf. Section III-B).

C. Experiments

1) k-means: *k*-means [40] is a vector quantization method originating from the field of signal processing. It is commonly used as an unsupervised clustering algorithm that is able to assign all observations, x, within a dataset into k disjoint partitions, each forming a cluster, (C_i). Formally, the method solves the problem

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{x \in C_i} dist(x, c_i) \tag{1}$$

given a distance metric $dist$ and each cluster centroid c_i. In practice, the Euclidean distance is often used as distance measure, effectively minimizing the inter-cluster variance. The *k*-means clustering problem is generally NP-hard, but can be efficiently approximated using an iterative optimization

[2]https://atmosphere.copernicus.eu/

[3]obtained from https://www.lanuv.nrw.de/umwelt/luft/immissionen/messorte-und-werte/

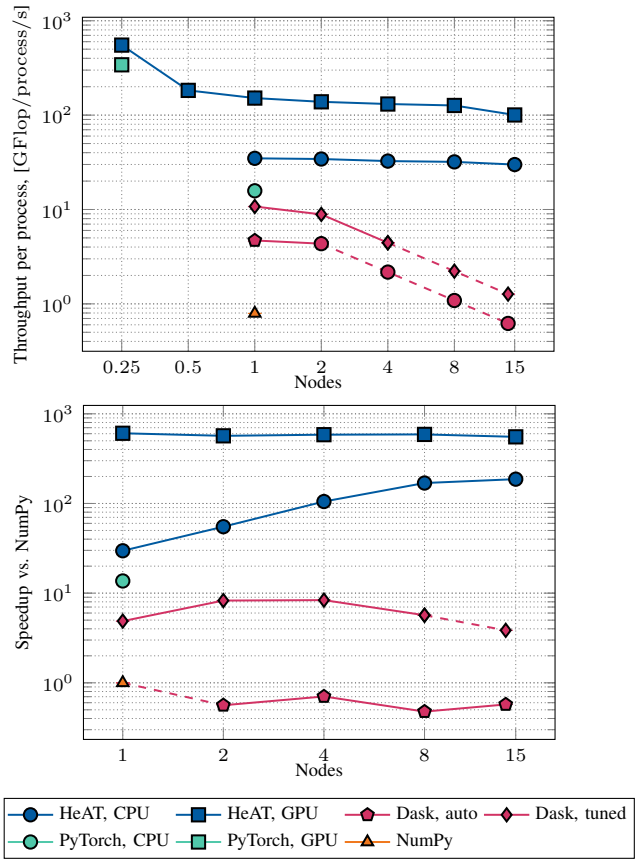


Fig. 4: *k*-means clustering: weak (*upper*) and strong scaling measurements (*lower*), cf. Section IV-C1.

method, such as, detecting a local minimum, i.e., Lloyd's algorithm [41]. In this experiment, we have benchmarked a *k*-means implementation relying on distance matrix computations between the data points and the centroids. These are dominated by element-wise vector operations in the distance matrix computation and reduction operations for finding the best matching centroids. For each benchmark, we have performed 30 optimization iterations at eight assumed centroids.

Weak scaling measurements are shown in Fig. 4 (upper panel). This shows that HeAT outperforms Dask by at least an order of magnitude. Furthermore, the weak scaling trend is nearly linear for HeAT, demonstrating solid scalability for larger datasets for both CPU and GPU. For Dask we were unable to complete the measurement procedure for all node configurations. While it was sporadically possible to complete the benchmark with a four-node configuration, it would terminate with an out-of-memory exception before the completion of the sequence. For 8 and 15 nodes, we were unable to obtain any measurements due to excessive memory consumption. Thus, the dashed line in the plots depicts a linear extrapolation of the previous trend while maintaining the temporal difference between the automatically chunked measurements and the tuned chunk size measurements.

For HeAT, a single GPU shows overall better performance compared to multiple GPUs (cf. Section IV-A). The difference in throughput between PyTorch and HeAT on a single node can be explained by a customized distance matrix computation implementation in HeAT. For both frameworks, we invoke the built-in `cdist` functions. However, the implementation in HeAT performs quadratic expansion of the Euclidean terms, instead of the naïve squaring and summation approach. The same implementation in PyTorch should result in similar performance.

Strong scaling measurements are shown in Fig. 4 (lower panel). Here, we obtain similar conclusions as in the weak scaling measurements. HeAT outperforms Dask by a significant margin and shows more favorable scaling behaviour. Again, we had to extrapolate two measurement points for Dask due to out-of-memory issues. While HeAT's CPU computations scale approximately linearly, the GPU back end shows strong linearity.

2) Forward-Propagation of a Neural Network: An objective of HeAT is the implementation of fully distributed neural networks. The first forward step of a neural network is composed of a matrix multiplication between the input data and a weight matrix and, if applicable, the addition of the relevant bias information. After this, an activation function is applied to the result. As the forward step of a network can be repeated many times, the performance differences of networks in different frameworks can increase drastically. HeAT's matrix multiplication is implemented using collective communications. This allows for the efficient usage of data by each process.

This benchmark is composed of the matrix multiplication of the input data (CityScapes) with a random weight matrix of size ($6\,291\,456 \times 128$), equivalent to a fully connected neural network layer with 128 neurons. The result of the multiplication is then fed through a rectified linear unit (ReLU) [42] activation function

$$max(XW, 0) \tag{2}$$

where X is the input data, W is the weight matrix, and $max()$ is the element-wise maximum of XW and 0.

The number of operations for this function is dominated by the matrix multiplication. Thus, weak scaling measurements should be nearly flat. As illustrated in Fig. 5 (upper panel), HeAT shows the expected behavior for both CPU and GPU calculations. On GPUs, HeAT maintains an average throughput of $(12.90 \pm 0.21)\,\mathrm{TFlop/s}$, or $(82.14 \pm 1.34)\,\%$ of the maximum [43]. Dask reaches a higher throughput on CPU at lower nodes due to its centralized scheduler, then performance degrades (cf. Section II).

The strong scaling measurements are shown in Fig. 5 (lower panel). While both HeAT and Dask show significant speedup against NumPy, HeAT shows higher speedups and more consistent scaling than Dask. The GPU measurements show a speedup of roughly a factor of two more than both

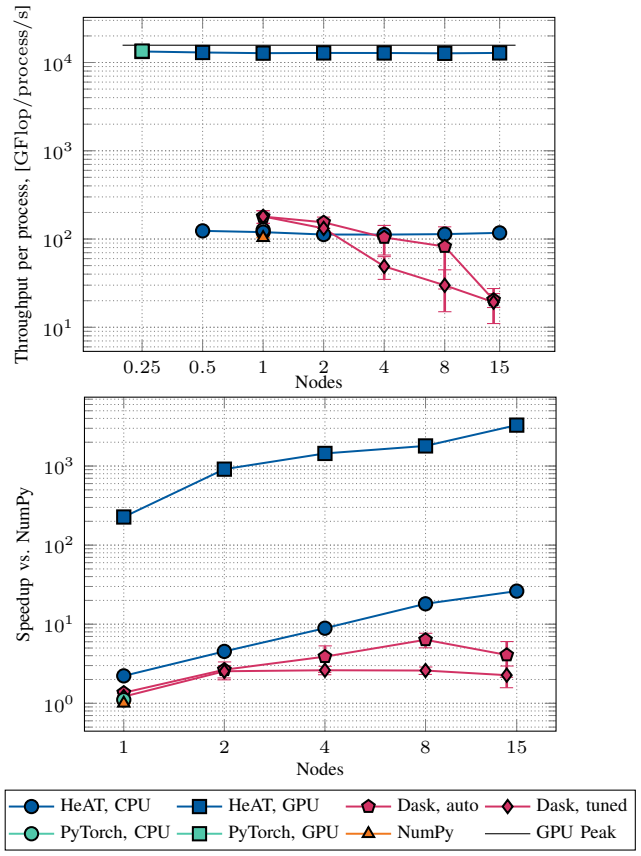


Fig. 5: Forward-propagation of a neural network: weak (*upper*) and strong scaling measurements (*lower*), cf. Section IV-C2.

Dask and NumPy.

3) LASSO: LASSO is a regression method of simultaneously applying regularization and parameter selection. Its basic form is an extension of the ordinary linear regression (OLS) method by introducing an L1-norm penalty of the parameters scaled by the regularization parameter. The corresponding objective function reads

$$E(w) = \|y - Xw\|_2^2 + \lambda \|w_-\|_1 \tag{3}$$

where y denotes the n samples of the output variables; $X \in \mathbb{R}^{n \times m}$ denotes the *system matrix* in which $m-1$ columns represent the different features, one column represents the constant bias term, and each of the n rows represents one data sample; $w \in \mathbb{R}^m$ denotes the regression coefficients; $w_- \in \mathbb{R}^{m-1}$ the regression coefficients of the features; and λ the regularization parameter.

In addition to the L2-norm regularization approach (i.e., Ridge-regression), LASSO favors not only smaller model parameters but, depending on the regularization parameters, can force selected model parameters to be zero. It is a popular method to determine the importance of input variables with respect to one or more dependent output variables.

In this experiment, a LASSO algorithm is used to determine the most important model parameters of the EURAD-IM model [35] on the forecast error of ozone. In order to minimize the objective function, a coordinate descent algorithm with a proximal gradient soft threshold applied to each coordinate was implemented in HeAT, Dask, NumPy, and PyTorch. For the weak scaling measurements, the LASSO algorithm is run for 20 iterations on a data sample size of 714 280 samples per node.

The HeAT CPU measurements show good weak scaling behaviour (Fig. 6, upper panel) with the highest throughput compared to the Dask and HeAT GPU versions. Dask shows poor weak scaling due to the incompleteness of Dask with respect to NumPy operations. For example, assignments to Dask arrays are not supported by the library itself but are heavily utilized in the implemented LASSO algorithm. Consequently, Dask cannot make efficient use of its lazy evaluation concept for this algorithm. The HeAT GPU version also does not scale well, albeit with a significantly higher throughput than Dask. This is due to the high number of communication operations required; the effect is increased when combined with a non-CUDA-aware MPI environment (cf. Section IV-A). Overall we can conclude that HeAT outperforms Dask by up to more than two magnitudes for weak scaling and by more than three magnitudes for strong scaling.

Strong scaling measurements (Fig. 6, lower panel) were conducted for the entire sample set. The trends observed in the weak scaling measurements are also visible here. Dask shows almost no scaling, whereas the HeAT CPU measurements indicate a good scaling behaviour. For the HeAT GPU implementation the speedup decreases with the increase in computing resources due to the non-CUDA-aware MPI environment.

4) Spectral Clustering: Spectral clustering is the process of partitioning data samples into k groups based on graph theory [44]. The underlying idea is to embed the dataset in the lower dimensional space of the k smallest eigenvalues of the graph's Laplacian matrix and then employ a clustering algorithm on them. The Laplacian matrix is derived from the adjacency matrix, which describes the edges, or links, between the data samples by a pairwise distance metric s_{ij}. The calculation of all pairwise metrics between n data samples has computational and memory complexity of $\mathcal{O}(n^2)$, where n is the number of data items. The subsequent eigenvalue decomposition of the Laplacian matrix is asymptotically in the same computational complexity class. The distributed spectral clustering algorithm in HeAT relies on calculating the similarity matrix $S = \{s_{ij}\}$ using ring-communication of data chunks. The eigenvalue decomposition is derived via the Lanczos algorithm [45], an adaptation of the power-method for finding extreme eigenvalues and corresponding eigenvectors. Unlike many other frameworks, our implementation calculates the exact similarity matrix, rather than an approximation via the Nystrom method [46]. Further, it does not require sparsification of this matrix, but can work on fully-connected

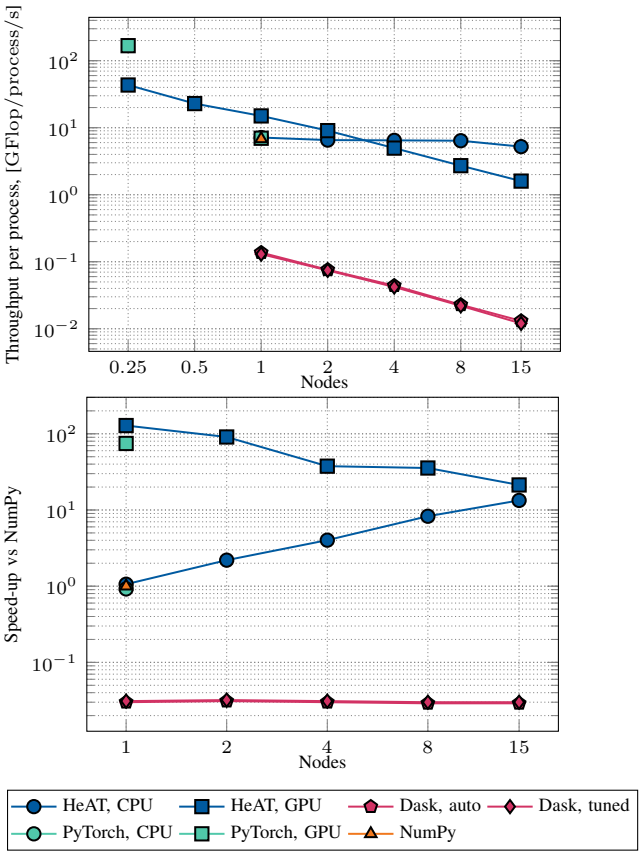


Fig. 6: LASSO regression: weak (*upper*) and strong scaling measurements (*lower*), cf. Section IV-C3

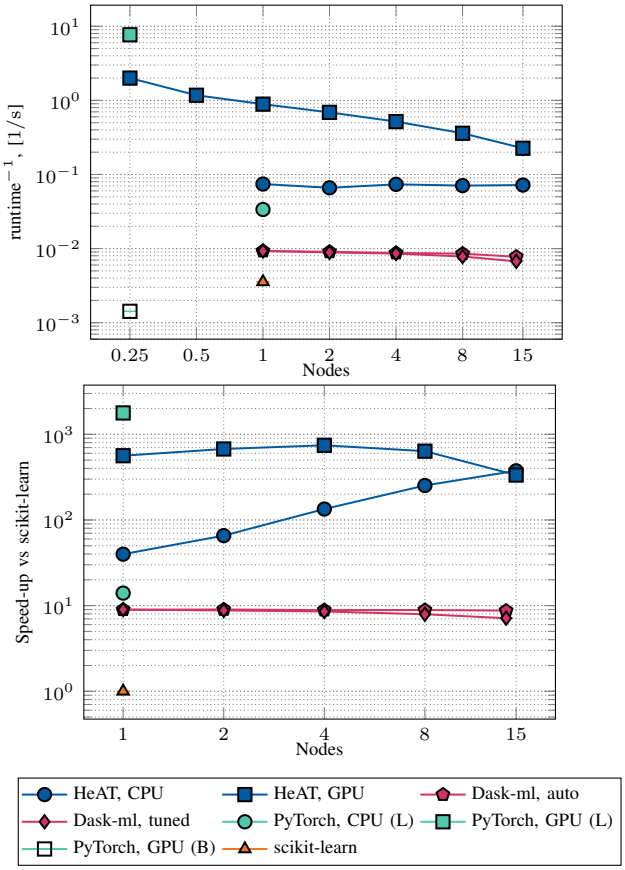


Fig. 7: Spectral clustering: weak (*upper*) and strong scaling measurements (*lower*), cf. Section IV-C4. Inverse run time is reported for weak scaling, as each framework utilizes a different implementation.

dense graph representations of the data.

The evaluated frameworks apply conceptually different workflows to perform spectral clustering. Dask-ml utilizes the Nystrom method for approximating the similarity matrix and its eigenvalues. Scikit-learn sparsifies the Laplacian matrix and applies the ARPACK library for eigenvalue decomposition. For PyTorch, spectral clustering was implemented for these experiments with both the built-in eigenvalue solver (B) as well as the Lanczos algorithm (L). Due to these differences in workflows, the comparison of the number of operations performed is not be meaningful. Thus, the inverse overall run time is reported.

Fig. 7 (upper panel) shows the results of the weak scaling experiment. HeAT clearly outperforms the other algorithms, except for the GPU version of the Lanczos implementation in PyTorch. However, with more processors, the communication overhead dominates computation time, primarily due to the plethora of pairwise distance calculations. On CPU, the PyTorch built-in implementation was canceled after exceeding the time limit. Results for the strong scaling measurements are presented in Fig. 7 (lower panel). On both CPU and GPU, the PyTorch built-in implementation measurements were not completed as they exceeded the time limit. For HeAT GPU measurements, the communication overhead causes an opposing trend to the increasing speedup at higher node numbers. Nonetheless, HeAT's spectral clustering algorithm provides superior run times to the other frameworks.

V. Discussion

We have presented HeAT, a Python-based framework for the distributed data analytics and machine learning on multiple CPUs and GPUs. It offers transparent parallel handling of data and operations to exploit the available hardware, be it personal workstations or world-class HPC systems. The NumPy-like API enables users to easily translate existing NumPy applications into distributed applications.

From the inception of HeAT, building parallelism into the dense linear algebra functions which compose the foundation of modern machine learning libraries has been a priority. PyTorch has been selected as an imperative, node-local compute engine for HeAT. As a direct result, HeAT benefits from PyTorch's highly optimized functions. However, distributed versions of these functions do not exist within PyTorch. HeAT is designed to alleviate this. To leverage the full potential of PyTorch, the algorithms within HeAT are designed based on a hierarchical strategy. For example, HeAT's

matrix multiplication utilizes PyTorch's matrix multiplication implementation on submatrices and aggregates the results via MPI communications. This hierarchical strategy enables the efficient utilization of the underlying hardware by exploiting locality in the memory hierarchy. An additional benefit of using PyTorch is that the associated run-time cost of Python is negligible because the computationally demanding functions are implemented in low-level languages.

Some of PyTorch's features do not directly map to HeAT. Among them is the just-in-time (JIT) compiler that enables the on-the-fly optimization of functions. While PyTorch's intermediate representation can only optimize its own operations and several native Python functions, it cannot optimize MPI routines. In order to make HeAT fully JIT-able, a low-level implementation of the MPI layer as a C++ PyTorch extension is necessary. Albeit work-intensive, it is a worthwhile opportunity for HeAT to obtain further performance gains.

For deep learning applications, several attempts have been made towards distributed model training (e.g. PyTorch `dist`, Horovod). These frameworks primarily focus on data-parallelism; however, generalized model parallelism is not available. HeAT's programming model facilitates straightforward data-parallelism as well as model parallelism and pipelining. The use of a custom communication layer allows for the implementation of distributed automatic differentiation, which is a vital part for a distributed model architecture.

In contrast to existing large-scale machine learning and data analytics frameworks, HeAT is designed with a focus on high-performance computing. Hence, significant efforts have been made to efficiently utilize the available hardware while avoiding central bottlenecks, such as workload schedulers and excessive I/O, e.g. serial file access or journaling. Although HeAT is unable to achieve peak hardware utilization in all cases, it enables the user to access a substantial portion of maximum performance with a high-level interface and the performance benefits are relatively independent of data characteristics.

Furthermore, the user-base of HPC resources is predominantly composed of domain experts who do not often have a strong background in parallelization. Hence, the number of configuration parameters for parallel constructs which are exposed to users should be minimized. At the same time, these constructs must be both powerful and versatile enough to allow for implementation of a large variety of distributed algorithms. HeAT's programming model offers a way to easily develop application-specific algorithms while leveraging the available computational resources, setting it apart from other approaches and libraries.

VI. Conclusions and Outlook

With HeAT, we address the needs of the ever-growing community of scientists, both in academia and industry, who seek to speed up the process of extracting information from large datasets. To this end, we have set upon the task of combining data analytics and machine learning algorithms with state-of-the-art high-performance computing into one, easy-to-use Python library.

The quality of an HPC machine learning library is defined by its performance. Section IV shows weak and strong scaling experiments on a number of applications: clustering (k-means, Section IV-C1, and spectral clustering, Section IV-C4); LASSO regression (Section IV-C3); and the forward-propagation step of a neural network (Section IV-C2). The presented results show that HeAT outperforms the most popular current competitor by up to two orders of magnitude in terms of processing speed.

In the short term perspective, the logical next step is the public availability of distributed automatic differentiation (cf. Section III-D2). This subsequently offers the opportunity for the development of high-level differentiable algorithms, such as data- and model-parallel neural networks. In light of the ever increasing need for machine learning models to yield reliable predictions, considerable efforts have been put towards the development of probabilistic approaches. HeAT's programming model and internal design give access to all levels of algorithmic development and by such offers an intuitive way to implement such approaches. One continuous objective also is the optimization of computation and memory performance of commonly used kernels.

We have demonstrated that even in its current early stage, HeAT offers great potential. The convergence of speed and usability sets it up to redefine high-performance data analytics by putting high levels of parallelism within easy grasp of scientists in academia and industry alike.

Acknowledgment

The authors would like to thank the system administrators at the Jülich Supercomputing Centre and in particular Dr. Alexandre Strube for their continuous support in maintaining the benchmarking HPC system. Furthermore, we want to thank the Helmholtz Analytics Framework collaboration for thorough feedback and valuable suggestions.

References

[1] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[2] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: a Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis,

J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org, [accessed at 2020-02-24]. [Online]. Available: http://tensorflow.org/

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[5] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling," in *Proceedings of the 14th Python in Science Conference (SciPy 2015)*, K. Huff and J. Bergstra, Eds., 2015, pp. 130–136.

[6] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS), 2015. [Online]. Available: https://fs.hlrs.de/projects/par/mpi//mpi31/

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," 2015, [accessed at 2020-02-02]. [Online]. Available: http://arxiv.org/abs/1512.01274

[9] A. Sergeev and M. Del Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," 2018, [accessed at 2020-03-17]. [Online]. Available: http://arxiv.org/abs/1802.05799

[10] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao, "Spartan: A Distributed Array Framework with Smart Tiling," in *Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association, 2015. ISBN 978-1-931971-22-5 p. 1–15. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/huang-chien-chin

[11] M. R. Kristensen, S. Lund, T. Blum, K. Skovhede, and B. Vinter, "Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster," in *Proceedings of the 4th Workshop on Python for High Performance and Scientific Computing (PyHPC13)*, 2013.

[12] M. Ravishankar and V. Grover, "Automatic acceleration of Numpy applications on GPUs and multicore CPUs," 2019, [accessed at 2020-03-17]. [Online]. Available: http://arxiv.org/abs/1901.03771

[13] S. Palkar, J. Thomas, D. Narayanan, A. Shanbhag, R. Palamuttam, H. Pirk, M. Schwarzkopf, S. Amarasinghe, S. Madden, and M. Zaharia, "Weld: Rethinking the Interface Between Data-Intensive Applications," 2017, [accessed at 2020-03-17]. [Online]. Available: http://arxiv.org/abs/1709.06416

[14] M. Bauer and M. Garland, "Legate NumPy: Accelerated and Distributed Array Computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019. doi: https://doi.org/10.1145/3295500.3356175 pp. 1–23.

[15] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018. ISBN 978-1-939133-08-3 pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz

[16] L. Chen, B. Peng, B. Zhang, T. Liu, Y. Zou, L. Jiang, R. Henschel, C. Stewart, Z. Zhang, E. Mccallum *et al.*, "Benchmarking Harp-DAAL: High Performance Hadoop on KNL clusters," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017. doi: https://doi.org/10.1109/CLOUD.2017.19 pp. 82–89.

[17] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimization Towards Training A Trillion Parameter Models," 2019, [accessed at 2020-03-17]. [Online]. Available: https://arxiv.org/abs/1910.02054

[18] IPython development team and Enthought, Inc., "DistArray: Think globally, act locally," 2020, [accessed at 2020-02-28]. [Online]. Available: http://docs.enthought.com/distarray

[19] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, "JAX: Composable Transformations of Python+NumPy Programs," 2018, [accessed at 2020-02-29]. [Online]. Available: http://github.com/google/jax

[20] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," 2018, [accessed at 2020-03-015]. [Online]. Available: http://arxiv.org/abs/1810.09679

[21] M. Merrill, W. Reus, and T. Neumann, "Arkouda: Interactive Data Exploration Backed by Chapel," in *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop (CHIUW 2019)*. ACM, 2019. doi: https://doi.org/10.1145/3329722. ISBN 9781450368001

[22] J. Daily and R. Lewis, "Using the Global Arrays Toolkit to Reimplement NumPy for Distributed Computation,"

in *Proceedings of the 10th Python in Science Conference (SciPy 2011)*, 2011. [Online]. Available: http://conference.scipy.org/scipy2011/slides/daily_GA_Toolkit.pdf

[23] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck *et al.*, "Asynchronous Execution of Python Code on Task-Based Runtime Systems," in *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2018, pp. 37–45.

[24] F. Darema, "The SPMD Model: Past, Present and Future," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis and J. Dongarra, Eds., vol. 2131, no. 1. Springer Berlin Heidelberg, 2001. doi: https://doi.org/10.1007/3-540-45417-9_1. ISBN 978-3-540-45417-5 pp. 1–1.

[25] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[26] L. Dagum and R. Menon, "OpenMP: an Industry Standard API for Shared-memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[27] C. Pheatt, "Intel® Threading Building Blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[29] L. Dalcín, R. Paz, M. Storti, and J. D'Elía, "MPI for Python: Performance Improvements and MPI-2 Extensions," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 655–662, 05 2008. [Online]. Available: https://doi.org/10.1016/j.jpdc.2007.09.005

[30] J. Salmon, M. Moraes, R. Dror, and D. Shaw, "Parallel Random Numbers: As Easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011. doi: https://doi.org/10.1145/2063384.2063405 pp. 1–12.

[31] N. K. Boiroju and K. Reddy, "Generation of Standard Normal Random Numbers," *Interstat*, vol. 5, pp. 1–14, 2012.

[32] D. Kundu and A. Manglick, "A Convenient Way Of Generating Normal Random Variables Using Generalized Exponential Distribution," *Journal of Modern Applied Statistical Methods*, vol. 5, no. 1, pp. 266–272, 2006.

[33] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[34] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016. doi: 10.1109/CVPR.2016.350. ISSN 1063-6919 pp. 3213–3223.

[35] H. Elbern, H. Schmidt, O. Talagrand, and A. Ebel, "4D-variational Data Assimilation with an Adjoint Air Quality Model for Emission Analysis," *Environmental Modelling & Software*, vol. 15, no. 6, pp. 539–548, 2000.

[36] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for Exotic Particles in High-energy Physics with Deep Learning," *Nature Communications*, vol. 5, p. 4308, 2014.

[37] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/SUSY

[38] The HDF Group, "Hierarchical Data Format, version 5," 1997, [accessed at 2020-02-29]. [Online]. Available: http://www.hdfgroup.org/HDF5/

[39] A. Collette, *Python and HDF5*. O'Reilly, 2013. ISBN 978-1449367831

[40] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, no. 14, 1967, pp. 281–297.

[41] S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[42] V. Nair and G. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on Machine Llearning (ICML-10)*. Omnipress, 2010. ISBN 9781605589077 pp. 807–814.

[43] *NVIDIA V100 Tensor Core GPU*, NVIDIA, 1 2020, [accessed at 2020-03-17]. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[44] U. Von Luxburg, "A Tutorial on Spectral Clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, 2007.

[45] C. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators," *Journal of Research of the National Bureau of Standards*, no. 45, pp. 255–282, 1950.

[46] C. Fowlkes, S. Belongie, F. Chung, and J. Malik, "Spectral Grouping Using the Nystrom Method," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 2, pp. 214–225, 2004.