Engineering a large-scale data analytics and array computing library for research: Heat

Fabian Hoppe[1], Juan Pedro Gutiérrez Hermosillo Muriedas[2], Michael Tarnawa[3], Philipp Knechtges[1], Björn Hagemeier[3], Kai Krajsek[3], Alexander Rüttgers[1], Markus Götz[2], Claudia Comito[3]

[1]German Aerospace Center (DLR), Institute of Software Technology, High-Performance Computing Department, Cologne (Germany), [2]Karlsruhe Institute for Technology (KIT), Scientific Computing Center (SCC), Karlsruhe (Germany), [3]Forschungszentrum Jülich GmbH (FZJ), Jülich Supercomputing Centre (JSC), Jülich (Germany),

Abstract: Heat is a Python library for massively-parallel and GPU-accelerated array computing and machine learning. It is developed *by* researchers *for* researchers, with the ultimate goal to make multi-dimensional array processing and machine learning for scientists (almost) as easy on a supercomputer as it is on a workstation with NumPy or scikit-learn. This paper highlights the relevance of this project to the research software engineering community by giving a short, but illustrative overview of Heat and discusses its role in the context of related libraries with a specific focus on its research software aspects.

Keywords: Multi-dimensional Arrays, Machine learning, Data Science, Data analytics, High-Performance Computing, Parallel Computing, GPUs, Big Data, Research Software

1 Introduction and overview

While the parallelization and scalability of deep learning applications on massive data have received considerable attention in recent years, scaling traditional numerical data analysis to large scientific datasets is still regarded as a niche activity for a few select areas of academic research, including earth observation, climate science and earth system modelling, astro- and particle physics, biophysics and -informatics, among others. These areas have traditionally been embedded in high-performance computing (HPC). Indeed, the challenge of adapting existing data analysis procedures to ever-increasing data volumes is a common experience in academia. The need to port applications from the workstation to the data center is a recognized and often insurmountable roadblock for most research groups across every field of science.

Nowadays, scientific data analysis workflows are typically implemented in Python, building on the highly versatile libraries **NumPy** [HMW+20], **SciPy** [VGO+20], and **scikit-learn** [PVG+11]. However, as researchers and/or research software engineers are confronted with ever-increasing data volumes, these foundational libraries are hampered by an essential drawback: they are limited to so-called shared-memory parallelization[1] and, consequently, to a single CPU of a workstation or cluster node; moreover, they do not support GPU acceleration. A

[1] A brief explanation of this and other parallelization-related terms is provided at the end of this introductory section.

research group seeking to overcome memory limitations (thereby accelerating their data analysis) might start by swapping memory-intensive algorithms for specialized solutions like machine learning (ML) or even deep learning (DL) algorithms to extract insights out of their large datasets. As data volumes increase, however, single-CPU memory bottlenecks emerge throughout the data processing chain, to the extent that even the necessary data preparation before ML or DL training becomes a challenge. We will go into the details of runtime and memory requirements for a sample data analysis pipeline on moderately large data in Sect. 4.

On this premise—that most research groups dealing with numerical data are, at one point or another, sitting on existing Python applications that can no longer be exploited because of the exploding data volumes— we develop and maintain **Heat**[2] [GDC+20]. This work started in 2018 within a project of the Helmholtz federation and is being continued as a cooperative effort of research groups from the authors' three host institutions. Heat is a generic Python library that facilitates scaling existing NumPy-/SciPy-/scikit-learn-based applications to high-performance computing (HPC) systems, including GPU clusters. More precisely, it implements an infrastructure of distributed-memory n-dimensional arrays, as well as data processing, linear algebra, and ML algorithms on top of that. The present paper does not replace the original Heat paper [GDC+20], mainly targeting (HPC-)experts. Instead, our aim is to point out the relevance of our work to a broader research software engineering (RSE) audience. Thus, the key contributions of this paper are as follows: First of all, we take a different, less technical, perspective than in [GDC+20] and put specific focus on Heat's research software character and the respective implications. Second, we provide a detailed discussion of related work that might also be of independent interest to the reader. Finally, our numerical experiments extend those of [GDC+20] as up-to-date versions of the respective software and modern hardware are used, memory requirements are tracked (in addition to runtimes), and a focus is placed on the necessity and realization of porting an originally non-HPC-capable code to HPC systems.

Structure of the paper

The remaining part of the paper is organized as follows: right at the end of this introductory section, we will give readers without an HPC background a brief overview of the most important parallelization- and HPC-specific terms used in the following. In Section 2 we discuss related work, i.e., we present an overview of the current state-of-the-art in distributed Python frameworks and discuss why we believe that Heat is filling an important gap in this field. Section 3 is devoted to Heat itself: in Subsection 3.1 we will examine in greater depth the features and design principles of our library; past and current applications are highlighted in Subsection 3.2. In Section 4 we illustrate our presentation by hands-on numerical examples: for a prototypical workflow consisting of some typical ML operations we first demonstrate the single-node limitations of scikit-learn; then, we address porting of the underlying code to Heat and (for comparison) Dask. On behalf of these examples we compare the respective numerical performance of Heat and Dask in terms of run time, memory consumption, and energy consumption. Finally, in the concluding Section 5 we provide an outlook on future development and applications of Heat.

[2] https://github.com/helmholtz-analytics/heat [Accessed May 03, 2024]

Frequently used terms

As announced, we provide a very short overview over some frequently used terms related to parallelization. A reader well familiar with these topics may thus directly continue with Section 2.

Shared-memory vs distributed-memory (parallelization) — Under shared-memory parallelism, each parallel process has access to the same, shared, memory, while in the distributed-memory setting each process can only access its own, private, memory. A typical example for a shared-memory environment is a multi-core CPU, whereas multiple nodes (i.e., machines) within an HPC-cluster form a distributed-memory environment. See, e.g., [VT24] for a brief introduction.

Single Program Multiple Data (SPMD) vs Multiple Program Multiple Data (MPMD) — These terms describe two different programming models for parallel computing, similar to Flynns taxonomy of parallel computer architectures [Fly66]. In the SPMD [Dar01] model each parallel process executes the same program, but with different underlying data (i.e., values of the variables); in contrast, in the MPMD model each parallel process can also execute a different program.

Task distribution / task-based parallelism — In this approach to parallelization, so-called tasks, i.e., "sequence[s] of instructions within a program that can be processed concurrently with other tasks in the same program" [TDH+18] are executed in parallel; see the reference for an extensive discussion.

Data distribution / data chunking — In a parallel setting, the entirety of data to be handled typically needs to be divided into pieces ("chunks") that are assigned to and/or accessed by different parallel processes. We refer to this in the following as data distribution (or chunking).

embarrassingly parallel [FWM94, Chapter 7.1] — This term describes workloads that can be easily parallelized with no (or sometimes only minimal) need for interaction (i.e., communication/synchronization) between the parallel processes.

2 Related work: Python's distributed-array ecosystem

The following discussion of related work considers libraries targeting *distributed array computing* in Python; "distributed" in this context refers at least to distributed onto several devices (e.g., multiple GPUs), with a clear focus on the memory-distributed setting, i.e. arrays distributed onto several machines (e.g., cluster nodes). Libraries and tools for accelerating or distributing Python code in general are not included if they do not explicitly support array computing. For a wider, more general overview, also covering this aspect, we refer the reader to the recent survey paper [CBST23]. The same applies to frameworks for distributed deep learning. In many cases, they support memory-distributed parallelization at least to some extent. We exemplarily mention the distributed-module of PyTorch[3], the DTensor[4] implementation, or general frameworks for distributed training such as **Horovod**[5] or **FairScale**[6]; nevertheless, if no array operations or classical machine learning algorithms are explicitly available in distributed mode, we do not discuss them further. The scientific data analytics library **ROOT**[7], developed at CERN, will also not

[3] https://pytorch.org/docs/stable/distributed.html [Accessed May 03, 2024]
[4] https://github.com/pytorch/pytorch/blob/main/torch/distributed/_tensor/README.md [Accessed May 03, 2024]
[5] https://github.com/horovod/horovod [Accessed May 03, 2024]
[6] https://github.com/facebookresearch/fairscale [Accessed May 03, 2024]
[7] https://root.cern/ [Accessed July 26, 2024]

be addressed further as it is heavily rooted in the C++ context (despite availability of Python bindings) and mainly targeting general C++ objects instead of arrays.

The necessity for a portable, generic Python framework to overcome memory bottlenecks and distribute NumPy/SciPy workflows has been a recognized issue for some time. Early efforts like **DistArray**[8] and **D2O**[9] [SGBE16] (both no longer maintained) implemented distributed *n*-dimensional arrays within a NumPy-like API and a limited set of features on multi-CPU systems already in the early 2010s. Currently, the landscape of distributed array computing and machine learning in Python is dominated by frameworks based on *task distribution* such as **Dask**[10] [Roc15] and, in the DL space, **Ray**[11] [MNW+18]. Dask distributes tasks using a centrally managed dynamic task scheduler. The scheduler process coordinates the actions of several Dask worker processes spread across multiple machines, and the concurrent requests of several clients. The workers execute the tasks and communicate with each other and the scheduler as needed. As such, Dask is a sophisticated and powerful tool to distribute Python-based array computing and machine learning tasks similar to NumPy and scikit-learn. GPU support however is only provided via third-party libraries (CuPy, cuNumeric, and cuML, with limitations, see below). Porting an existing NumPy/SciPy/scikit-learn code to Dask can be quite complex, as the programming style slightly differs (see examples in Section 4, particularly Listing 4), and the size and shape of the data chunks must often be tuned by the user via trial and error; in particular, it needs to be ensured that the RAM of a worker is sufficient for the tasks to be performed on it. Naturally, understanding memory consumption under this approach, i.e. distributing tasks in MPMD style, is slightly more intricate than under the more traditional (in HPC) approach to distribute data in SPMD style, which is applied for Heat. Moreover, the task scheduler model introduces significant overhead: our benchmarks in Section 4 indicate that Heat's memory-distributed, communication-optimized algorithms are able to outperform Dask significantly for certain operations in terms of runtime, and especially memory consumption. **Xorbits**[12] [LHQ+24], formerly Mars[13], is roughly comparable to Dask in its overall approach and scope.

Equally pressing in the scientific and data-science community is the wish to speed up existing NumPy/SciPy/scikit-learn applications by enabling (multi-)GPU computing. In the CUDA ecosystem built around Nvidia GPUs, **CuPy** [OUN+17] is intended to serve as a straight-forward, GPU-ready drop-in replacement for NumPy, however distributed, i.e. multi-GPU, capabilities are quite limited to a few operations[14] like min, max or sum; matrix-matrix-multiplication is the only supported linear algebra operation so far. **cuNumeric**[15], developed by Nvidia on top of Legate [BG19], enables GPU acceleration for NumPy arrays. However, as of early 2024, it does not enable parallel I/O from shared memory, which significantly limits its usability in the context of memory-intensive operations that would require loading large data from a file. **cuML**[16],

[8] https://github.com/enthought/distarray [Accessed May 03, 2024]
[9] https://gitlab.mpcdf.mpg.de/ift/D2O [Accessed May 03, 2024]
[10] https://github.com/Dask/Dask [Accessed May 03, 2024]
[11] https://github.com/ray-project/ray [Accessed May 03, 2024]
[12] https://github.com/xorbitsai/xorbits [Accessed December 11, 2024]
[13] https://github.com/mars-project/mars [Accessed December 11, 2024]
[14] https://docs.cupy.dev/en/stable/reference/generated/cupyx.distributed.array.DistributedArray.html [Accessed May 13, 2024]
[15] https://github.com/nv-legate/cunumeric [Accessed May 03, 2024]
[16] https://github.com/rapidsai/cuml [Accessed May 03, 2024]

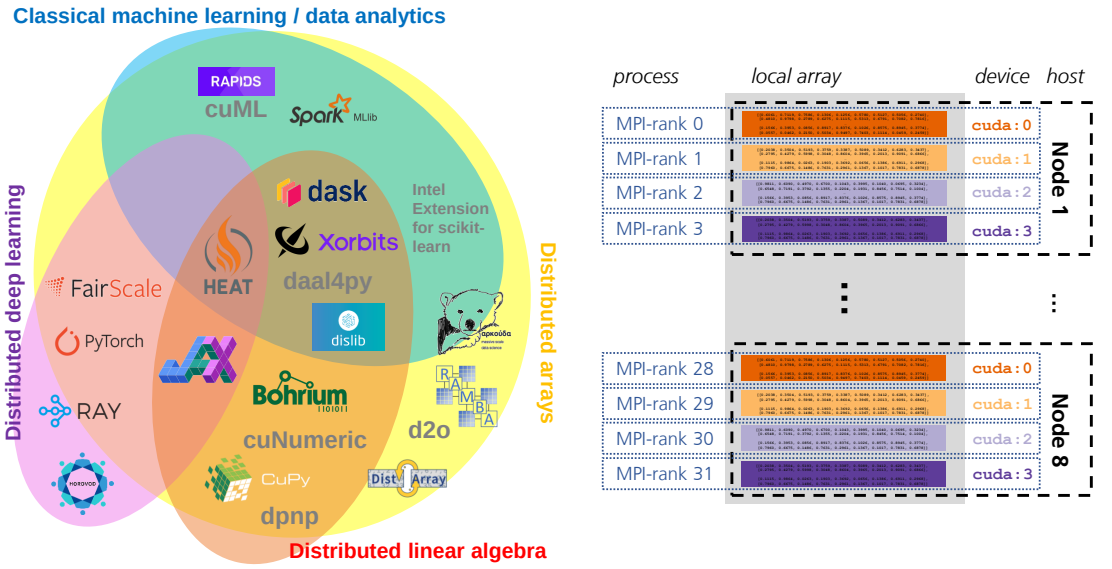

Figure 1: *Left hand side:* The landscape of distributed array computing and machine learning in Python w.r.t. the scope of the libraries. *Right hand side:* Illustration of a 2-dimensional `DNDarray` on GPU that is distributed over 8 nodes / hosts with 4 GPUs each ("`cuda:0`" - "`cuda:3`") and 64 MPI-processes (i.e. one GPU per MPI-process).

offers CUDA-accelerated drop-in replacements for scikit-learn functions; multi-node execution is mostly based on Dask.

Apart from the Nvidia ecosystem, an increasingly popular and rapidly evolving option to distribute and accelerate NumPy functions, is the Google library **Jax**[17] [FJL18]; the focus of the library is to enable hardware acceleration (on GPUs of different vendors and Google TPUs) as well as straight-forward SPMD-parallelization and automatic differentiation (AD) of all operations. Jax however follows the paradigm of functional programming which is quite different from NumPy's rather object-oriented approach. As a consequence, Jax only parallelizes pure functions, i.e., functions whose outputs are based only on their inputs and do not modify anything outside of themselves in the process, and Jax arrays are immutable, unlike NumPy or Heat arrays; therefore, e.g., assigning a new value to a `numpy.ndarray` or to its distributed counterpart in Heat (see Subsection 3.1 for a description) at a given index is easy, whereas the same operation requires a functional workaround in Jax. While DL applications can be easily built with the help of several libraries on top of Jax, we are not aware of a scikit-learn-like library in the Jax-ecosystem.

A number of other libraries populate specialized niches of the Python distributed-array ecosystem. Intel develops **Ramba**[18] [PAM22], speeding up and parallelizing NumPy operations on CPU with minimal changes using compiled functions on process, and a Ray or MPI-based backend for distribution. So far Ramba only implements a fraction of the features that are already

[17] https://github.com/google/jax [Accessed May 03, 2024]
[18] https://github.com/Python-for-HPC/ramba [Accessed May 03, 2024]

available to Heat users, and does not support GPUs, while **dpnp**[19] offers drop-in replacements for certain NumPy functions that can be executed efficiently on Intel GPUs. Other options like **MLlib**[20] based on Apache Spark, **Arkouda**[21] [MRN19] based on the Chapel programming language, **dislib**[22] [CS19] based on PyCOMPSs [TBA+17], are not easily portable across HPC systems.

In our view, a Python library for high-performance scientific large-scale data analysis ideally must satisfy the following conditions:

- *portability* from personal laptop to cluster systems, ideally to cloud resources, supporting diverse operating systems and hardware;
- *interoperability* within the Python array and ML ecosystem (i.e., NumPy, SciPy, scikit-learn, PyTorch etc.);
- *multi-node parallelism* enabled for all operations (including those that are not embarrassingly parallel), and transparent to the user;
- *multi-GPU acceleration* across vendor ecosystems;
- *scalability* of operations and algorithms and efficient usage of available HPC resources;
- *ease of use* via a standardized, intuitive API;
- *research-focused user support* to accommodate scientists' unique needs in terms of features, software quality and reliability / reproducibility, as well as project and publication timelines.

Table 1 (at the very end of the paper) and Figure 1 (lhs) summarize our overview of the landscape of parallel array computing in Python, focusing on the central aspects: multi-node capabilities, GPU-acceleration, a simple NumPy-/scikit-learn-like API, interoperability / portability (in terms of both software and hardware requirements), as well as scope (from basic array operations to linear algebra and classical machine learning, up to deep learning).

3 Heat

3.1 Heat in a nutshell

A detailed technical description of Heat and its programming model can be found in the original Heat publication [GDC+20]. Therefore, in this section, we will summarize Heat's design and features for a research software engineering (RSE) audience. Heat strives to satisfy all of the conditions we regard as important for this kind of software; see the end of the previous section. To ensure this, Heat's development is guided by the following five main paradigms, the first four being functional design requirements, the fifth one serving as reminder where Heat comes from and what it is developed for:

[19] https://github.com/IntelPython/dpnp [Accessed May 03, 2024]
[20] https://spark.apache.org/docs/latest/api/python/index.html [Accessed May 03, 2024]
[21] https://github.com/Bears-R-Us/arkouda [Accessed May 03, 2024]
[22] https://github.com/bsc-wdc/dislib [Accessed May 03, 2024]

Multi-node parallelism — Heat allows for memory-distributed, i.e., multi-node / multi-host, parallelization and is not limited to embarrassingly parallel applications in this setting. Heat distributes data, not tasks, and thus allows the user to take full advantage of the memory-distributed setting when dealing with memory-intensive use-cases.

Hardware acceleration — Heat can take advantage of modern GPUs (both Nvidia and AMD); nevertheless, CPUs —including multi-threading— are natively supported as well. The handling of devices (CPU or GPU) is as straightforward as in PyTorch.

Simple API and usage — Heat has a simple NumPy-/scikit-learn-like API [BLB+13] that allows for rapid prototyping as well as for easy adaptation of existing codes and workflows. Applications can be easily run with the `mpirun` command.

Platform independence / Interoperability — Heat is based on PyTorch and MPI and thus avoids restrictions to very specific hardware or software environments as much as possible. In particular, both Nvidia and AMD GPUs, and Intel, AMD, and Arm-based CPUs are supported so far.

Scientific background — Heat is developed as a general-purpose HPC array computing and machine learning library by scientists for scientists; in particular the FAIR4RS principles [BCK+22] are complied with.

The last point hereof deserves particular attention as it also heavily influences the preceding points. As it is developed for a research purpose, Heat clearly satisfies the common, prescriptive definition of *"research software"*, used, e.g., by the FAIR4RS working group [BCK+22]. Since the current use is exclusively within research, there is a paper about the software [GDC+20], there are research results [Bli, Bou, NHG+22] obtained with this software and even a publication on such results [DRP+20], the software is developed by (academic) researchers, and has been cited by researchers (outside the developers team) [ALP+24, DCB+23, AIS+23, HC21, RPVS21], Heat also meets some stricter criteria for "research software" (in distinction to *"software in research"*) discussed in [GKL+21]. Except for dislib—which is developed in academia as well—Heat is, as far as we know, the only currently maintained and developed library in the ecosystem discussed above that also meets these stricter criteria and thus may indisputably be identified as "research software"; for at least some of the competitors developed outside academia, the distinction from "software in research" under the stricter criteria may be considered unclear.

In order to comply with the FAIR ("Findable, Accessible, Interoperable, Reusable") principles of research software development [BCK+22], we provide an open repository (MIT licence) hosted on GitHub, were everyone can submit related issues or feature requests, while also periodically releasing to the Python Package Index to conform to the standard way to provide python packages; furthermore, a docker file is provided for those users who require containerization. Each release is accompanied with a DOI in Zenodo, to ensure each version of Heat is citable. To ensure correct functionality and reproducibility, each change to the code base goes through multiple steps of quality assurance, including code reviews, automated static code analysis, con-

tinuous testing[23], on multiple hardware[24] and software stacks, and continuous benchmarking using the performance monitoring tool **perun**[25] [GFD+23] to detect performance degradation.

Heat arrays are compatible with other popular array computing libraries, giving it a high degree of interoperability; in particular, interoperability with PyTorch is given by construction, whereas interoperability with NumPy is a central design goal. Heat's main dependencies are the Python libraries PyTorch and mpi4py. **PyTorch**[26] [PGM+19] is a large, open-source tensor framework for deep learning. We make use of its highly optimized and versatile implementation of single-process *n*-dimensional arrays ("`torch.Tensor`") and corresponding array computing routines as process-local compute engine; in particular, Heat inherits the support both for Nvidia and AMD GPUs as well as the ability to utilize multi-threading on multi-core CPUs from PyTorch. Memory-distributed parallelism is built on top of PyTorch utilizing **mpi4py**[27] [DPSD08], which offers Python-wrappers for the C++-implementation of the MPI-standard (*"Message Passing Interface"*) [Mes15]. Since PyTorch is widely used and supports all hardware architectures common in HPC, and MPI is the de facto standard for memory-distributed parallelism in traditional HPC, these dependencies are very unlikely to present users with problems. This ensures a high degree of interoperability and is the reason why we label Heat as "platform-independent" in Figure 1 (lhs) and Table 1.

For convenience of the reader, we now provide a summary on Heat's data object, the distributed *n*-dimensional array ("`DNDarray`") class, its programming model, and the respective implications; for details we refer again to [GDC+20]. Figure 1 (rhs) illustrates the basic idea of this class on behalf of a two-dimensional `DNDarray` that is distributed over 32 MPI-processes on 8 nodes with 4 GPUs each. The data of the entire, global array are split along the rows and distributed over the available MPI processes; each process has only direct access to its own fraction of the entire number of rows. In our example, each such process is associated with exactly one GPU and the local data, given by a PyTorch tensor, are stored on this device, here called "`cuda:0`" - "`cuda:3`" by PyTorch. This has the advantage that –for this concrete example– the array does not need to fit into the memory of a single GPU, but rather into the combined memory of all GPUs. Non-trivial operations on such an array will require the processes to exchange data. Heat implements an MPI communication layer (via mpi4py) and the necessary data exchange for all operations where needed, while the actual process-local computations are implemented in PyTorch.

Technically speaking, Heat follows the SPMD paradigm of parallel computing as each MPI-process executes the same Python program, but with different underlying data. Thus, the overall style may be described as *hybrid-parallel and bulk-synchronous* since process-local, shared-memory parallel computations (utilizing PyTorch's native OpenMP-, CUDA-, or ROCm-support) alternate with MPI-based, memory-distributed inter-process communication, the latter including also potentially synchronizing / blocking communication operations. Moreover, so-called *"eager execution"* is done, i.e., operations are performed immediately when they appear in the program

[23] current code coverage by unit tests is about $\approx 92\%$ [accessed May 04,2024]
[24] currently on hardware with an Nvidia- and AMD-GPU, respectively, provided on codebase.helmholtz.cloud by HIFIS (Helmholtz Digital Services for Science) of the Helmholtz Association of German Research Centres
[25] https://github.com/Helmholtz-AI-Energy/perun [Accessed May 03, 2024]
[26] https://pytorch.org/, https://github.com/pytorch/pytorch [Accessed May 03, 2024]
[27] https://github.com/mpi4py/mpi4py [Accessed May 03, 2024]

code; this is in contrast to "lazy execution", where certain operations are not executed immediately, but only when their results are needed for further calculations. In terms of transparency of resource usage of a certain operation as well as in terms of easy prototyping and debugging, we consider eager execution as advantageous compared to lazy execution.

Heat's programming model implies that, on a multi-process architecture, each process (each node or each GPU) initially reads load-balanced slices of the input data directly from shared memory. From that moment on, any data exchange or synchronization that may be necessary for the execution of operations is conducted via point-to-point or collective MPI calls between processes. Array copies are avoided whenever possible, in line with NumPy guidelines. A memory-intensive operation is distributed across the entire available resources—certainly with some overhead if inter-node communication is required—and does not need to be broken down manually in operations on smaller arrays by the user. For scientific data processing and analysis the implication is huge, as coherent, interdependent, but massive data units can still be processed as units, albeit memory-distributed, and do not need to be broken up in pseudo-independent chunks. Porting existing NumPy/SciPy/scikit-learn code to Heat is straightforward: typically, the user simply needs to specify the dimension along which the input data will be sliced, and if desired, the device. A concrete example will be provided in the Section 4.

3.2 Past and current applications of Heat

Before moving on to concrete examples in the next section, we will briefly highlight two applications of Heat in scientific research. An outlook to currently planned future applications will be provided in the final Section 5.

3.2.1 "Rocket science" with Heat

In a study at the German Aerospace Center (DLR), researchers used Heat to analyze datasets from hybrid rocket combustion experiments. Each dataset comprised 30,000 high-resolution images captured from high-speed video footage, recording dynamic combustion processes over three seconds at 10,000 frames per second.

The primary objective was to employ clustering techniques to discern distinct combustion phases and identify transient phenomena that are critical for optimizing rocket engine performance. To achieve this, two different clustering algorithms implemented in Heat were used: K-Means clustering and spectral clustering. K-Means clustering was chosen for its efficiency on large datasets, allowing researchers to quickly group similar combustion states based on visual similarity [RPK20]. Spectral clustering was applied to further refine these classifications by considering the connectivity of data points in a lower-dimensional space, thus identifying more complex patterns and especially short-term turbulent structures that K-Means might overlook [DRP+20].

3.2.2 Post-processing in earth system modelling with Heat

Heat is currently used in production by a research group at the Institute of Bio- and Geosciences (IBG) within the Forschungszentrum Jülich (FZJ). In this concrete application, a post-processing

workflow for outputs of the ParFlow[28] hydrological model makes use of Heat as its computational backend; hereby, Heat's multi-node, multi-GPU-capabilities enable and accelerate the processing of the big data volumes produced from the ensemble-based simulations. This post-processing tool, ParFlow Diagnostics[29], is openly available on GitHub. Associated research results on, e.g., continental-scale high-resolution soil moisture reanalysis have been presented in [NHG+22]. Moreover, the master's thesis [Bou] has been written in this context.

4 Hands-on

To give the reader a more illustrative impression of memory limitations in the context of scikit-learn ML applications, and to demonstrate the Heat user experience, we show how typical code snippets in the context of array computations and machine learning written in NumPy and/or scikit-learn can be ported to a cluster via Heat, and compare this with the respective steps in Dask. The comparison with Dask is motivated by the fact that this library is likely the most widely used among those mentioned in Section 2 at the moment. As our test data set we utilize the ATLAS Top Tagging Open Data Set [ATL22], a data set with roughly 42 million data points with 819 features each; represented in single-precision floating-point format, this corresponds to roughly 135 GB of data. We consider the following operations that are typical components of data analytics workflows: loading the data from the .h5-file[30], stacking the data to a matrix (some of the features are saved as separate data sets in the .h5-file), pre-processing by applying in-place standardization, computing the truncated SVD of the data (with prescribed accuracy), K-means Clustering with appropriate initialization (fit and predict), and linear regression with Lasso-regularization (fit and predict).

Our numerical experiments have been performed on the following HPC-systems: DLRs terrabyte cluster[31] for experiments on GPUs and DLRs cluster CARO[32] for experiments on CPUs; low-level software is loaded from the module system of the respective cluster, while Heat, Dask, and scikit-learn together with their high-level dependencies are installed in Python virtual environments, respectively; see the above footnotes for the corresponding details.

[28] https://www.parflow.org/ [Accessed May 29, 2024]

[29] https://github.com/HPSCTerrSys/ParFlowDiagnostics [Accessed May 29, 2024]

[30] HDF5 [For98] is an HPC-compatible data format common for storing large scientific data sets; see also https://www.hdfgroup.org/ [Accessed May 29, 2024].

[31] The system is operated by LRZ (Munich). Its GPU-nodes are equipped with 2 Intel Xeon Gold 6336Y 24 cores 185 W 2.4 GHz, 1024 GB RAM, and 4 Nvidia HGX A100 80 GB 500 W GPUs. Software loaded as modules: Python 3.10.10, OpenMPI 4.1.5 (Intel compiler 2023.1.0), CUDA 11.8, SLURM 21.08.8-2. Software installed via pip in a virtual environment: Heat 1.4.1, NumPy 1.26.3, h5py 3.11.0, PyTorch 2.2.0+cu118, mpi4py 3.1.6, perun 0.6.2.

[32] The system is operated by GWDG (Göttingen). Its CPU-nodes are equipped with 2 AMD EPYC 7702 64 cores 200 W 2.0 GHz each and 256 GB RAM ("medium" partition) or 1000 GB RAM ("bigmem" partition). Software loaded as modules: Python 3.9.16, OpenMPI 4.1.5 (GCC 10.4.0), SLURM 23.11.4. Software installed via pip in a virtual environment: Heat 1.4.1, dask 2024.4.2, dask_ml 2024.4.4, dask_mpi 2022.4.0,NumPy 1.26.4, h5py 3.11.0, PyTorch 2.2.2, mpi4py 3.1.6, perun 0.6.2., scikit-learn 1.4.6

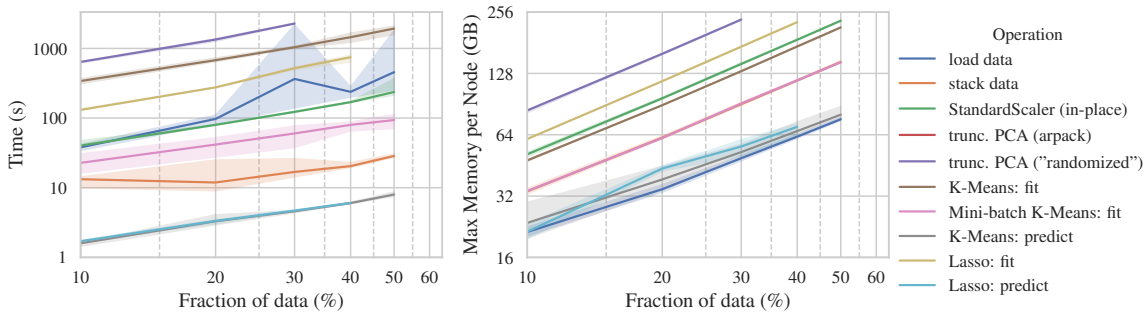

Figure 2: Memory- and runtime consumption for each of the operations using scikit-learn. The lines indicate averages, the shaded areas the range of observations over 5 runs.

4.1 Implementation in scikit-learn

First of all, let us consider an implementation in scikit-learn (see Listing 1) and the associated results on 64 CPU-cores of a single "medium" node of DLRs cluster CARO. Because of single-node memory limitations affecting NumPy, SciPy and scikit-learn, none of these operations can be successfully performed on the entire data set. Accordingly, we show runtime and memory consumption results on representative fractions of the entire data set. We observed that pre-processing, K-Means clustering, and Lasso could only be performed on 60 %, 50 %, and 40 % of the data, respectively, until an out-of-memory error appeared; in some cases this seems to be due to scikit-learn falling back to some double precision computations internally although the data set has been loaded as single precision. The truncated SVD with prescribed accuracy could not even be computed for 10 % of the data (as the underlying full SVD in NumPy resulted in an `init_gesdd`-error); the alternative solvers for scikit-learns PCA-routine resulted in an LAPACK integer overflow error (for the "arpack" solver) or, again, an out-of-memory for 30 % or more of the data (for the "randomized" solver). Switching to MiniBatchKMeans (with the suggested batch size of 256×64) could avoid the out-of-memory error only for 40 % or less of the data. Maximum memory consumption and runtime for each of the operations are shown in Figure 2. Memory and time requirements grow at least proportional to the amount of data to be processed, respectively. It becomes evident that already for scientific data set with moderate size, such as the present one, the RAM limit of a single machine may pose a severe limitation.

The fact that NumPy, SciPy, and scikit-learn are literally running out of memory when it comes to the processing of really huge data sets, underlines the need to enable a *distributed-memory* setting, i.e., being able to perform computations on several machines in parallel, e.g., on several nodes of a cluster, hence exploiting the cumulative RAM of all machines together. Taking into account hardware acceleration even increases the urgency of using memory-distributed alternatives, as up to now the RAM available for GPUs usually is significantly smaller than that available for CPUs.

4.2 From scikit-learn to Heat or Dask

In the following, we exemplify porting our scikit-learn example from Listing 1 to memory-distributed execution on multi-CPU, multi-GPU architecture via Heat, and for comparison also with Dask.

```
import numpy as np
import h5py
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.linear_model import Lasso

filename = <...>

# load data from file
with h5py.File(filename,'r') as file:
    features = file.keys()
    arrays = [np.asarray(file[feature]) for feature in features if (feature
    ↪  != 'labels' and feature != 'weights')]
    labels = np.asarray(file['labels'])
    weights = np.asarray(file['weights'])

# stack data to matrix
for k in range(len(arrays)):
    arrays[k] = arrays[k].reshape(n_data_points,-1)
data = np.hstack(arraylist)

# pre-processing
scaler = StandardScaler(copy=False)
scaler.fit_transform(data)

# PCA with prescribed tolerance
rtol = 2.5e-2
pca = PCA(n_components=1-rtol)
data_pca = pca.fit_transform(data)

# K-Means
kmeans = KMeans(n_clusters=25)
kmeans.fit(data)
labels = kmeans.predict(data)

# Linear regression by Lasso
lasso = Lasso(alpha=0.1, max_iter=100)
X = weights.reshape(-1,1)**0.5 * data
y = weights**0.5*2*(labels-0.5)
lasso.fit(X, y)
prediction = lasso.predict(data)
```

Listing 1: Typical operations in a machine learning workflow implemented in scikit-learn

```
import h5py
import heat as ht
```

```
3
4  filename = ...
5
6  # load data
7  features = h5py.File(filename,'r').keys()
8  arrays =
   ↪  [ht.load_hdf5(filename,feature,split=0,device="gpu",dtype=ht.float32)
   ↪  for feature in features if ...]
9   ...
10
11 # stack data to matrix
12 data = ht.hstack(arrays)
13
14 # pre-processing
15 scaler = ht.preprocessing.StandardScaler(copy=False)
16 scaler.fit_transform(data)
17
18 # PCA with prescribed tolerance
19 rtol = 2.5e-2
20 data_pca = ht.linalg.hsvd_rtol(data,rtol)
21
22 # K-Means
23 kmeans = ht.cluster.KMeans(n_clusters=25, init="batchparallel")
24 kmeans.fit(data)
25 labels = kmeans.predict(data)
26
27 # Linear regression by Lasso
28 lasso = ht.regression.Lasso(0.1, max_iter=100, tol=1e-4)
29 X = weights.reshape(-1,1)**0.5 * data
30 y = weights**0.5*2*(labels-0.5)
31 lasso.fit(X, y)
32 prediction = lasso.predict(data)
```

Listing 2: Heat implementation of the machine learning workflow

Listing 2 show the corresponding workflow implemented in Heat. Note that the changes required for the transition from scikit-learn to Heat are marginal for, e.g., stacking, K-Means, and Lasso; for `StandardScaler` or the arithmetics in lines 38/39 of Listing 1 literally no changes are necessary at all. A scikit-learn-similar API for PCA is planned for the near future. The most important, but still rather simple, change happens for loading the data: the argument `split=0` indicates that the resulting `DNDarray` is supposed to be split along the axis with number 0, i.e. along the axis enumerating the 42 million data points in our case. The argument `device="gpu"` specifies that the array is loaded to, stored, and processed in the GPU-memory; omitting this argument yields the fallback default option `device="cpu"`. Hence, switching from CPU to GPU is as straightforwards in Heat as known from PyTorch.

Let us point out that the code in Listing 2 does not contain any explicit reference to parallelization or data distribution except for specifying the "`split`"-axis along which the data shall be distributed over the available computational resources. If all dependencies are available, the script can be run using the standard `srun`- or `mpirun`-commands on a HPC-system (with SLURM as scheduler) or —mainly for development and debugging, of course— on a workstation or notebook; see Listing 3.

```
srun --nodes=8 --ntasks-per-node=4 --gres=gpu:4 python ml_pipeline.py
OMP_NUM_THREADS=16 MKL_NUM_THREADS=16 srun --nodes=8 --ntasks-per-node=8
 ↪  --cores-per-task=16 python ml_pipeline.py
mpirun -n 4 python ml_pipeline.py
```

Listing 3: Command line instructions for running the Heat implementation on 8 GPU-nodes with 4 GPUs each on an HPC-system with SLURM (first line), on 8 CPU-nodes with 8 MPI-processes per node and 16 cores per process (second line), or simply on a workstation or notebook with 4 CPU-cores (third line). For the second line, `device="gpu"` needs to be replaced by `device="cpu"`, of course. In the third line, if `device="gpu"` in the python script and a GPU is available, the GPU will be used.

We now compare this with our implementation of the same workflow using Dask (cf. Listing 4); at this point, it should be noted that the authors of this paper are not Dask experts and —despite their best efforts— the proposed implementation may not be the best possible. Let us just note that —in our opinion— the handling of data distribution (called "chunking" in Dask) together with lazy evaluation is somewhat more complicated than in Heat as the differences from NumPy / scikit-learn are slightly larger. To make data distribution into distributed memory actually happen before the computations (which is advantageous for their performance), e.g., `client.persist`-calls as in line 34 are necessary.

```
import os
import h5py
import numpy as np
import dask
import dask.array as da
from dask_mpi import initialize
from distributed import Client
from dask_ml.preprocessing import StandardScaler
from dask_ml.cluster import KMeans
from dask_ml.decomposition import PCA
from dask_ml.linear_model import LinearRegression


filename = ...
num_procs = int(os.getenv('SLURM_NTASK'))
num_threads = int(os.getenv('SLURM_CPUS_PER_TASK'))

# initialize Dask client
initialize(interface="ib0", nthreads=num_threads)
client = Client()

file = h5py.File(filename, mode='r')
features = list(h5py.File(filename,'r').keys())

# prepare chunk sizes
global_length_along_0 = f[features[0]].shape[0]
chunk_length_along_0 = int(global_length_along_0/(num_procs-1))
chunksizes = {key: np.array(f[key].shape) for key in features}
for key in features:
```

```
    chunksizes[key][0]=chunk_length_along_0
    chunksizes[key]=tuple(chunksizes[key])

# load data from file
xs = [client.persist(da.from_array(file[feature][:,...],
 ↪  chunks=chunksizes[feature])) for feature in features if (feature !=
 ↪  'labels' and feature != 'weights')]

# stack data to matrix
for k in range(len(xs)):
    xs[k] = da.reshape(xs[k], (loadLength, -1))
x = da.concatenate(xs, axis=1)
x = da.rechunk(x, chunks=(chunkLength, x.shape[1]))
x = client.persist(x)

# pre-processing
x = x.astype(float)
scaler = StandardScaler(copy=False)
x = scaler.fit_transform(x)

# PCA (randomized)
pca = PCA(n_components=691,svd_solver="randomized")
data_pca_rand = pca.fit(x)

# PCA (full and truncated)
pca = PCA(n_components=691,svd_solver="full")
data_pca_full = pca.fit(x)

# K-Means
clusterer = KMeans(n_clusters=25, init="k-means||")
clusterer.fit(x)
y = clusterer.predict(x)

labels, weights = ... # similar as above

# Linear regression by Lasso
X = client.persist(weights.reshape(-1,1)**0.5 * x)
y = client.persist(weights**0.5*2*(labels-0.5))
lasso = LinearRegression(penalty="l1", tol=1e-2, C=0.1, max_iter=100)
lasso.fit(x,y)
y = client.persist(lasso.predict(X))

file.close()
client.shutdown()
```

Listing 4: Dask implementation of the machine learning workflow

4.3 Heat vs Dask: Runtime, memory and energy consumption

We investigate performance and scalability of the respective implementations in Heat (Listing 2) and Dask (Listing 4) in terms of runtime as well as memory and energy consumption. Since our focus regarding the performance is on scalability, we investigate the so-called weak scal-

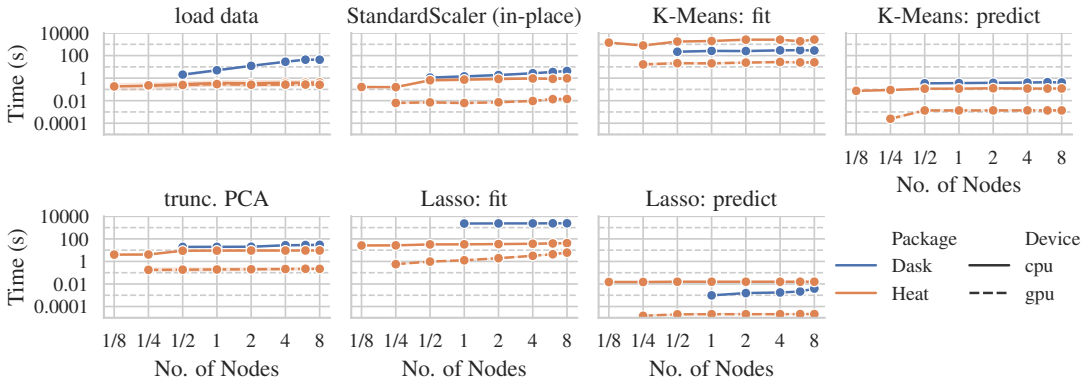

Figure 3: Runtime of different typical ML operations in Heat (CPU and GPU) and Dask (CPU). The (dashed) lines show the average runtime and the shaded areas indicate the range of observed runtimes each. (Smaller values and less growth are better.)

ing behaviour, i.e., we increase the amount of data proportional to the compute resources until the entire data set is processed on the maximum resources (here: 8 CPU- or GPU-nodes, respectively). Thus, in the Heat experiments each MPI-process receives about 650 000 data points (CPU) or 1.3×10^6 data points (GPU). The experiments have been conducted on CPU- (Heat and Dask) and GPU-nodes (Heat only) with 4 MPI-processes per node (i.e., 1 GPU per MPI-process) for the GPU-experiments and 8 MPI-processes per node (i.e., 16 CPU-cores per MPI-process) for the CPU-experiments; consequently, in the Heat experiments 1/2 etc. node refers to 2 or 4 MPI-processes etc. on a GPU- or CPU-node, respectively. Since for Dask, one MPI-process is reserved for the scheduler anyway, we run our experiments with Dask with at least 2 MPI-processes (1/4 node); the experiments for 2 processes, however, did not complete within the 24 h time limit which is the reason that the results start with 1/2 node (4 MPI-processes).

Due to time constraints, the complete workflow was run only once; after that, the workflow was run 10 times without the most time-consuming operation (K-Means fit and predict for Heat, Lasso fit and predict for Dask), respectively, in order to obtain statistically more meaningful results for at least a subset of the operations. Let us also point out that we consider the comparability of K-Means and Lasso in Heat and Dask to be rather weak, since the initialization of K-Means and the entire algorithm of Lasso are very different in both libraries. Similarly, the approaches for computing an approximate truncated PCA are different (randomized SVD in Dask and hierarchical SVD in Heat) although we expect the results of both methods to be of comparable quality.

In Figure 3, it can be seen that Heat outperforms Dask in terms of runtime for loading data, standardization, truncated SVD, Lasso fit, and K-means predict, while Dask is significantly faster than Heat for stacking the data, Lasso predict, and K-Means fit. Due to lazy evaluation in Dask, however, we are not absolutely sure whether for stacking and Lasso predict actual computations took place in Dask, whereas regarding K-Means the difference in the runtimes is likely due to the implementation in Heat currently being less efficient from an algorithmic point of view than in Dask. In the cases where Heat performs better than Dask, we believe this is due to the

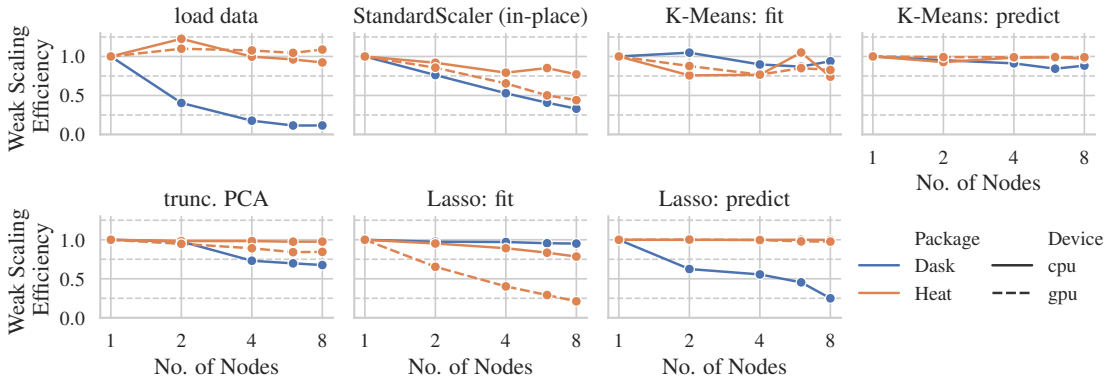

Figure 4: Weak scaling efficiency of typical ML operations in Heat (CPU and GPU) and Dask (CPU). The (dashed) lines show the geometric mean of the observed values. (Higher values are better.)

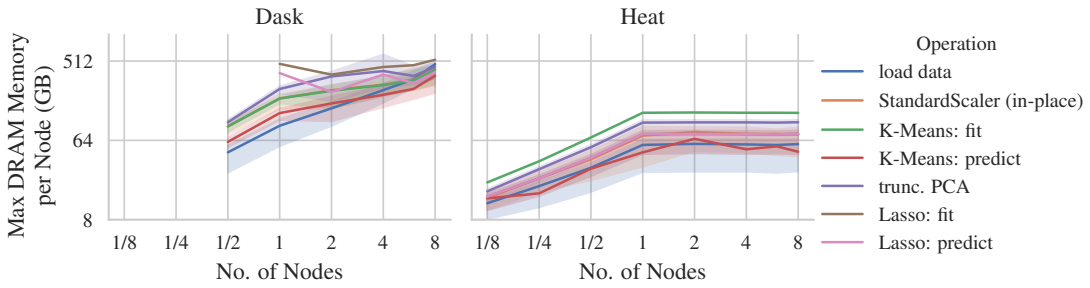

Figure 5: Memory consumption for typical ML operations of Heat and Dask on CPU. The lines show the averages and the dashed areas the range of observed values, respectively. (Smaller values and less growth beyond one node are better.)

overhead associated with Dask's centrally managed dynamic task scheduling. In terms of scaling efficiency (of the runtime), the results shown in Figure 4 indicate that Heat scales similarly or better than Dask for most of the operations, although there is an undeniable uncertainty due to the comparatively large variations between the rounds.

The situation is much clearer when considering the maximum memory consumption of the different operations; see Figure 5 (lhs). Dask consumes significantly more memory than Heat for all operations and configurations. In particular, the Dask experiments had to be run on the "bigmen" partition of CARO, while all Heat experiments could be run on the default "medium" partition. It can be clearly seen that the amount of RAM used by Heat *per node* remains almost constant for the experiments on 1 to 8 nodes, while it keeps increasing for Dask; this might eventually become problematic when considering a more data-intensive use case. We suspect that the observed differences in memory requirements can be explained by the fact that Heat by design distributes data, while Dask primarily distributes tasks.

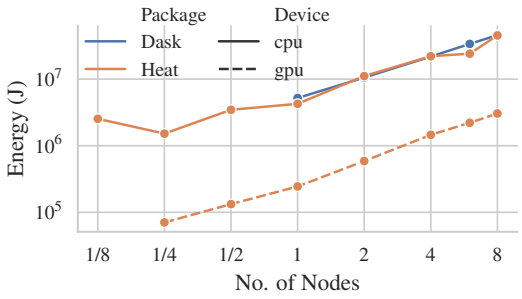

Figure 6: Energy consumption for a single run of the workflow (all operations) on CPU (Heat and Dask) and GPU (Heat only).

Finally, as awareness of the need to use energy responsibly is increasing in particular in the context of machine learning and AI [DPS+23], we report the energy consumption for the entire workflow, i.e. for the first run only. The results shown in Figure 6 indicate that in certain cases it can also make sense to use GPUs from an energy-efficiency point of view, as their increased energy consumption (compared to CPUs) can be offset by a significantly lower runtime.

5 Future challenges, development, and applications

To begin the discussion of current and future challenges, let us briefly categorise our software within the realm of research software in general. In doing so, we follow an approach for structuring the latter from [HDB+24]: on the role-based scale, Heat itself falls mainly into the third category (*"Research Infrastructure Software"*); its intention is to be used within the first category (containing software for data analytics in the context of a specific domain-level research question). Regarding the developer-based categorization, the project currently falls into the *"Project Group"*-category as three groups from different host institutions collaborate on Heat's development; however, with the participation at Google Summer of Code 2022[33] we made first steps to expand towards the "Community"-category. On the maturity-based scale, the most part of Heat falls into the third category (*"Accepted Methods and Models"*) as our intention is to provide the user with rather established and reliable instead of experimental algorithms. Nevertheless, from time to time a partial overlap with the second category (*"Novel Methods and Models"*) comes into play since our work sometimes requires adapting and modifying existing algorithms as well as benchmarking and comparing them. Finally, on the level of application classes in institutional software engineering guidelines, Heat is grouped in "Application class 2" as we aim at long-term development and maintenance.

Another popular approach for the characterization of software created and used in research is the concept of the *software stack* [Hin19] that identifies six layers ranging from hardware (the lowest) to project-specific code (the highest). Here, one may locate Heat in one of the middle to upper layers, probably best in the *"Scientific infrastructure"* layer. Herein, the distinction

[33] https://summerofcode.withgoogle.com/archive/2022/organizations/forschungszentrum-julich [Accessed May 02, 2024]

from the next lower layer "Non-scientific infrastructure" —containing, e.g., compilers or programming languages— is obvious, whereas there might be a partial overlap with the next upper layer "Domain-specific tools" as further development is often driven by the needs of a specific application.

The fact that our software is by design in the infrastructure category and mainly falls into the "project group" category in terms of the development team is currently the biggest non-scientific challenge in further development. Despite welcome progress in recognising software as a research output, the focus of most institutional and third-party funders remains on domain-specific outputs rather than underlying infrastructure. In other words: in the worst-case scenario, the work on Heat is too methodical to be funded as concrete specialised research, but too applied to count as pure methodological research. Funding pure maintenance, i.e. classic software engineering tasks such as updating dependencies, proper releases etc., is even more difficult in a scientific context. The challenge is therefore to find an appropriate balance between further development in the context of concrete domain-specific applications (with a rather narrow focus prescribed by the needs of the application), further development with a broader methodological focus (i.e., with a view to the library as a whole, in particular with regard to re-usability) and the maintenance of the already existing code. Despite these drawbacks, from our personal perspective as developers and scientists, working at the intersection of methodological research and concrete application is quite appealing and varied.

After the first five years of Heat development, our main objective at present is to make the transition to long-term development and maintenance, which is usually particularly challenging in the context of research software as the early deaths of similar projects show. To make this transition, we are considering particularly the following two measures: first, we want to increase the number of users (especially outside our host institutions), and second, we want to take active steps towards more community-based development. To achieve the latter, participation in community-building efforts such as the currently forming *"High Performance Software Foundation"*[34] may be a good opportunity that we are currently discussing. To achieve the first goal, we are actively promoting our software in various scientific communities and offering help to scientists who are considering trying out Heat for their research; such help may range from support during installation, hints on adapting existing workflows up to the implementation of new features required for a particular application. Especially the last point should be considered as valuable for scientists and distinguishes Heat from similar libraries. We thus conclude the paper with an outlook on ongoing and planned future efforts in this direction:

Climate change and resilience: Heat for anomaly detection in earth observation data

Heat is going to be applied at DLR within an ongoing DLR project on risks for coastal regions in the context of climate change and associated resilience measures[35]. The task to be performed with Heat is anomaly detection in a large dataset of Earth observation images of the Northern and Baltic Seas. This dataset, spanning from 2016 to 2023 and comprising a total of 5 TB, includes remote sensing images collected every six days. Anomaly detection will be performed using the so-called Local Outlier Factor (LOF) algorithm multiple times, which requires the computa-

[34] https://hpsf.io/ [Accessed May 02, 2024]
[35] https://www.dlr.de/en/pi/research-transfer/projects/resikoast [Accessed May 02, 2024]

tion of a huge number of pairwise Euclidean distances (in the order of $\mathcal{O}(10^9)$) between image details [RP21]. The respective part of the project team and the Heat developers are currently investigating how Heat can be used to handle the associated computational demands efficiently, e.g., using Heat's implementation of `cdist` or a possibly improved or adapted version of it; in particular, we expect a new type of distance computation (based on a structural similarity index measure (SSIM)) and a scalable implementation of LOF to be added as new features to Heat as part of this collaboration.

Space science and beyond: Heat for ESA applications

In the course of the recently started, ESA-funded (European Space Agency) early technology development project ESAPCA *"Enabling the analysis of extremely large data sets by scalable and hardware-accelerated PCA and DMD"*[36] Heat will be made fit for various ESA applications by adding massively-parallel implementations of SVD, PCA, and DMD (Dynamic Mode Decomposition) as new features to Heat. These methods are indispensable and ubiquitous in data science and engineering, but computationally challenging in the context of large data sets as runtime and memory footprint can easily grow superlinearly as a function of data size. The intended application areas at ESA include the data-driven modelling of thermospheric density and problems arising in the area of digital manufacturing, e.g., in situ measurements of powder bed solidification. In the context of this project, we also want to explore the opportunities of using Heat in the data science workflow of the new-space startup *parametry.ai*. As the idea of transfer is becoming increasingly important in all areas of science we regard this as an important step for future development as well.

Matter and light: massive data processing in astro-/particle physics

We support the radioastronomical and nuclear physics communities. In both cases, researchers look to Heat to adapt existing NumPy/SciPy applications to exploding data volumes, the former to detect and flag radiofrequency interferences in their data, the latter for particle tracking in high occupancy detectors with machine learning algorithms on GPUs.

Acknowledgements: The authors gratefully acknowledge the computational resources provided through the joint high-performance data analytics (HPDA) project "terrabyte" of the German Aerospace Center (DLR) and the Leibniz Supercomputing Center (LRZ). The authors gratefully acknowledge the scientific support and HPC resources provided by the German Aerospace Center (DLR). The HPC system CARO is partially funded by "Ministry of Science and Culture of Lower Saxony" and Federal Ministry for Economic Affairs and Climate Action". This work is supported by the Helmholtz project HiRSE_PS and Helmholtz AI. Moreover, the authors thank all past and current contributors to Heat for their valuable work.

[36] https://activities.esa.int/4000144045 [Accessed May 02, 2024]

Bibliography

[AIS+23] M. Aach, E. Inanc, R. Sarma, M. Riedel, A. Lintermann. Large scale performance analysis of distributed deep learning frameworks for convolutional neural networks. *Journal of Big Data* 10(1):96, Jun 2023.
doi:10.1186/s40537-023-00765-w

[ALP+24] C. Alt, M. Lanser, J. Plewinski, A. Janki, A. Klawonn, H. Köstler, M. Selzer, U. Rüde. A continuous benchmarking infrastructure for high-performance computing applications. *International Journal of Parallel, Emergent and Distributed Systems* 39(4):501–523, 2024.
doi:10.1080/17445760.2024.2360190

[ATL22] ATLAS Collaboration. ATLAS Top Tagging Open Data Set. 2022.
doi:10.7483/OPENDATA.ATLAS.FG5F.96GA

[BCK+22] M. Barker, N. P. Chue Hong, D. S. Katz, A.-L. Lamprecht, C. Martinez-Ortiz, F. Psomopoulos, J. Harrow, L. J. Castro, M. Gruenpeter, P. A. Martinez, T. Honeyman. Introducing the FAIR Principles for research software. *Scientific Data* 9(1):622, Oct. 2022. Number: 1 Publisher: Nature Publishing Group.
doi:10.1038/s41597-022-01710-x

[BG19] M. Bauer, M. Garland. 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.
doi:10.1145/3295500.3356175

[BLB+13] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. Pp. 108–122. 2013.

[Bli] L. Blind. Parallel Dynamic Mode Decomposition - Identifying spatiotemporal patterns with HPC. Bachelor Thesis, Fachhochschule Aachen, Campus Jülich, 2021.
doi:10.5281/zenodo.7682668

[Bou] B. Bourgart. Massiv parallele Datenanalyse für die Erdsystemmodellierung mit dem Helmholtz Analytics Toolkit. Bachelor Thesis, Fachhochschule Aachen, Campus Jülich, 2019.
https://juser.fz-juelich.de/record/866456

[CBST23] O. Castro, P. Bruneau, J.-S. Sottet, D. Torregrossa. Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications. *ACM Comput. Surv.* 56(3), oct 2023.
doi:10.1145/3617588

[CDG+22] D. Coquelin, C. Debus, M. Götz, F. von der Lehr, J. Kahn, M. Siggel, A. Streit. Accelerating neural network training with distributed asynchronous and selective optimization (DASO). *Journal of Big Data* 9(1):14, 2 2022.
doi:10.1186/s40537-021-00556-1

[CS19] J. Álvarez Cid-Fuentes, S. Solà, P. Álvarez, A. Castro-Ginard, R. M. Badia. dislib: Large Scale High Performance Machine Learning in Python. In *Proceedings of the 15th International Conference on eScience*. Pp. 96–105. 2019.

[Dar01] F. Darema. The SPMD Model: Past, Present and Future. In Cotronis and Dongarra (eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Lecture Notes in Computer Science 2131(1), pp. 1–1. Springer Berlin Heidelberg, 2001.
doi:10.1007/3-540-45417-9_1

[DCB+23] D. Degen, D. Caviedes Voullième, S. Buiter, H.-J. Hendricks Franssen, H. Vereecken, A. González-Nicolás, F. Wellmann. Perspectives of physics-based machine learning strategies for geoscientific applications governed by partial differential equations. *Geoscientific Model Development* 16(24):7375–7409, 2023.
doi:10.5194/gmd-16-7375-2023

[DPS+23] C. Debus, M. Piraud, A. Streit, F. Theis, M. Götz. Reporting electricity consumption is essential for sustainable AI. *Nature Machine Intelligence* 5(11):1176–1178, Nov 2023.
doi:10.1038/s42256-023-00750-1

[DPSD08] L. Dalcín, R. Paz, M. Storti, J. D'Elía. MPI for Python: Performance Improvements and MPI-2 Extensions. *Journal of Parallel and Distributed Computing* 68:655–662, 05 2008.
doi:10.1016/j.jpdc.2007.09.005

[DRP+20] C. Debus, A. Rüttgers, A. Petrarolo, M. Kobald, M. Siggel. *High-performance data analytics of hybrid rocket fuel combustion data using different machine learning approaches*. 2020.
doi:10.2514/6.2020-1161

[FJL18] R. Frostig, M. J. Johnson, C. Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4(9), 2018.
https://mlsys.org/Conferences/doc/2018/146.pdf

[Fly66] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE* 54(12):1901–1909, 1966.
doi:10.1109/PROC.1966.5273

[For98] B. Fortner. HDF: The hierarchical data format. *Dr Dobb's J Software Tools Prof Program* 23(5):42, 1998.

[FWM94] G. Fox, R. Williams, P. Messina. *Parallel Computing Works!* Parallel processing scientific computing. Morgan Kaufmann, 1994.

[GDC+20] M. Götz, C. Debus, D. Coquelin, K. Krajsek, C. Comito, P. Knechtges, B. Hagemeier, M. Tarnawa, S. Hanselmann, M. Siggel, A. Basermann, A. Streit. HeAT – a Distributed and GPU-accelerated Tensor Framework for Data Analytics. In *2020 IEEE International Conference on Big Data (Big Data)*. Pp. 276–287. 2020.
doi:10.1109/BigData50022.2020.9378050

[GFD+23] J. P. Gutiérrez Hermosillo Muriedas, K. Flügel, C. Debus, H. Obermaier, A. Streit, M. Götz. perun: Benchmarking Energy Consumption of High-Performance Computing Applications. In Cano et al. (eds.), *Euro-Par 2023: Parallel Processing*. Pp. 17–31. Springer Nature Switzerland, Cham, 2023.
doi:10.1007/978-3-031-39698-4_2

[GKL+21] M. Gruenpeter, D. S. Katz, A.-L. Lamprecht, T. Honeyman, D. Garijo, A. Struck, A. Niehues, P. A. Martinez, L. J. Castro, T. Rabemanantsoa, N. P. Chue Hong, C. Martinez-Ortiz, L. Sesink, M. Liffers, A. C. Fouilloux, C. Erdmann, S. Peroni, P. Martinez Lavanchy, I. Todorov, M. Sinha. Defining Research Software: a controversial discussion. Dec. 2021.
doi:10.5281/zenodo.5504016

[HC21] K. M. A. Hasan, S. Chakraborty. GPU Accelerated Tensor Computation of Hadamard Product for Machine Learning Applications. In *2021 International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD)*. Pp. 1–5. 2021.
doi:10.1109/ICICT4SD50815.2021.9396980

[HDB+24] W. Hasselbring, S. Druskat, J. Bernoth, P. Betker, M. Felderer, S. Ferenz, A.-L. Lamprecht, J. Linxweiler, B. Rumpe. Toward Research Software Categories. *Preprint on arXiv*, 2024.
doi:10.48550/arXiv.2404.14364

[Hin19] K. Hinsen. Dealing With Software Collapse. *Computing in Science & Engineering* 21(3):104–108, 2019.
doi:10.1109/MCSE.2019.2900945

[HMW+20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. Array programming with NumPy. *Nature* 585(7825):357–362, Sept. 2020.
doi:10.1038/s41586-020-2649-2

[LHQ+24] W. Lu, K. He, X. Qin, C. Li, Z. Wang, T. Yuan, X. Liao, F. Zhang, Y. Chen, X. Du. Xorbits: Automating Operator Tiling for Distributed Data Science. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. Pp. 5211–5223. May 2024.
doi:10.1109/ICDE60146.2024.00392

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS), 2015.
https://fs.hlrs.de/projects/par/mpi//mpi31/

[MNW+18] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Pp. 561–577. USENIX Association, 2018.
https://www.usenix.org/conference/osdi18/presentation/moritz

[MRN19] M. Merrill, W. Reus, 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:10.1145/3329722

[NHG+22] B. Naz, H.-J. Hendricks-Franssen, K. Görgen, B. Bourgart, C. Montzka, C. Comito, D. Coquelin, S. Kollet. An ensemble-based parallel data assimilation and data analytics framework for the development of continental-scale high-resolution soil moisture reanalysis. 5 2022.
https://juser.fz-juelich.de/record/917267

[OUN+17] R. Okuta, Y. Unno, D. Nishino, S. Hido, C. Loomis. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017.
http://learningsys.org/nips17/assets/papers/paper_16.pdf

[PAM22] B. Pillai, T. A. Anderson, T. Mattson. Ramba: High-performance Distributed Arrays in Python. In *Proceedings of the 21th Python in Science Conference*. 2022.
https://www.scipy2022.scipy.org/posters

[PGM+19] 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, S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Pp. 8024–8035. Curran Associates, Inc., 2019.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12:2825–2830, 2011.
http://jmlr.org/papers/v12/pedregosa11a.html

[Roc15] M. Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In Huff and Bergstra (eds.), *Proceedings of the 14th Python in Science Conference (SciPy 2015)*. Pp. 130–136. 2015.
https://conference.scipy.org/proceedings/scipy2015/pdfs/matthew_rocklin.pdf

[RP21] A. Rüttgers, A. Petrarolo. Local anomaly detection in hybrid rocket combustion tests. *Experiments in Fluids* 62(136):1–16, 2021.
doi:10.1007/s00348-021-03236-1

[RPK20] A. Rüttgers, A. Petrarolo, M. Kobald. Clustering of paraffin-based hybrid rocket fuels combustion data. *Experiments in Fluids* 61(1):1–17, 2020.
doi:10.1007/s00348-019-2837-8

[RPVS21] N. A. Rink, A. Paszke, D. Vytiniotis, G. S. Schmid. Memory-efficient array redistribution through portable collective communication. *Preprint on arXiv*, 2021.
doi:10.48550/arXiv.2112.01075

[SGBE16] T. Steininger, M. Greiner, F. Beaujean, T. Enßlin. d2o: a distributed data object for parallel high-performance computing in Python. *Journal of Big Data* 3, 2016.
doi:10.1186/s40537-016-0052-5

[TBA+17] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, J. Labarta. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications* 31(1):66–82, 2017.
doi:10.1177/1094342015594678

[TDH+18] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, D. S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74(4):1422–1434, Apr 2018.
doi:10.1007/s11227-018-2238-4

[VGO+20] 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. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, 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, SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17:261–272, 2020.
doi:10.1038/s41592-019-0686-2

[VT24] M. Verdicchio, C. Teijeiro Barjas. *Introduction to High-Performance Computing*. Pp. 15–29. Springer US, New York, NY, 2024.
doi:10.1007/978-1-0716-3449-3_2

Name	**by**	**lifetime**	**Intended scope**				API	m-n	GPU	idp.	**Comments**
		(latest release)	DA	LA	ML	DL					
Arkouda	M. Merrill / W. Reus (US DoD)	2019 - present (v2024.04.19)	✓				✓	✓		*	*builds on Chapel
Bohrium	University of Copenhagen	2011 - 2020 (v0.11.0)	✓	✓			✓	✓	✓	✓	
RAPIDS / cuML	Nvidia	2019 - present (v24.04.00)			✓			✓*	✓	**	*multi-node, multi-GPU mode is based on Dask **built on CUDA and thus only support for NvidiaCUDA-GPUs
Legate / cuNumeric	Nvidia	2021 - present (v23.11.00)	✓	✓			✓	(✓)*	✓	**	*No scalable I/O routine available **requires CUDA and thus supports Nvidia-GPUs only
CuPy	Preferred Networks / community	2017 - present (v13.1.0)	✓	✓			✓	*	✓	(✓)**	*module cupyx.distributed (based on NCCL) only contains matmul as a linalg functionality, most operations on distributed array are not implemented yet **focus seems to be on Nvidia-CUDA-GPUs as support of AMD-ROCm-GPUs is "experimental" as of v13.0.0
D2O	Max Planck Institute for Astrophysics	2016 - 2017 (no release)	(✓)								
daal4py	Intel	2019 - 2020* (v2024.3.0)			✓			✓	**	***	*Drop-in functionality for scikit-learn has been deprecated and moved to Intel(R) Extension for scikit-learn **GPU-support has been deprecated ***no ARM-based CPUs
Dask	M. Rocklin / community	2015 - present (v2024.4.2)	✓	✓	✓			✓*	✓	(✓)**	*Dasks task-based parallelization approach ("worker-scheduler") allows for multi-node parallelization as long as the RAM of each node is sufficient for the respective tasks. **Dask arrays on GPU need to be realized via CuPy.
dislib	Barcelona Supercomputing Center	2018 - present (v0.9.0)	✓	✓	✓			✓	✓	*	*parallelization builds on the task-based framework PyCOMPSs, GPU-support is built on CuPy
DistArray	NASA / Enthought	2008 - 2015 (v0.6)	✓				✓	✓		✓	
dpnp	Intel	2020 - present (v0.14.0)	✓	✓			✓	✓	✓	?*	*hardware requirements not stated explicitly in the documentation; a focus on or even restriction to Intel-hardware is likely
Heat	DLR, JSC, KIT	2018 - present (v1.4.1)	✓	✓	✓	(✓)*	✓	✓	✓	✓	*data-parallel training of neural networks as in the torch.distributed package as well as the DASO-algorithm [CDG+22]
Intel Extension for scikit-learn	Intel	2021 - present (v2024.3.0)			✓			*	✓	**	*currently no distributed mode **no ARM-CPUs; direct support of Intel-GPUs only as usage of Nvidia-CUDA- or AMD-ROCm-GPUs requires a dedicated plug-in (the latter being in beta-state for AMD-devices)
Jax	Google	2020 - present (v0.4.26)	✓	✓		✓*	✓	✓	✓**	✓	Jax arrays are always immutable (unlike NumPy) and Jax is aimed to work best with functional programming (see Castro et al, Table 3). *several libraries for DL based on Jax are available **Jax natively also supports Google TPUs.
Xorbits (Mars)	Xorbits Inc. (Alibaba)	2018 - present (v0.7.2)	✓	✓	✓		✓	✓	✓	(✓)*	*GPU-support is based on CuPy
PySpark / MLlib	Apache Software Foundation	2013 - present (v3.5.1)			✓			✓	(✓)*	**	*Nvidia-GPUs can be used for certain algorithms via Spark Rapids ML based on cuML **MLlib is Apache Spark's scalable machine learning library and thus relies on Spark
Ramba	Intel	2021 - present (pypi: v0.1.post157)	✓	✓			✓	✓		✓	

Table 1: Overview of the libraries and their features. "DA", "LA", "ML", and "DL" stand for "basic distributed arrays", "distributed linear algebra", "distributed (classical) machine learning", and "distributed deep learning", respectively. "API", "m-n", "GPU", and "idp." are abbreviations for "Simple NumPy-/scikit-learn-like API", "multi-node capabilitites", "GPU-support", "platform independence". The "lifetime" of a project refers to the span between the first and the latest release on the corresponding repository; projects with the latest release dating back more than one year (i.e., prior than April 2023) are considered "dead".