The Helmholtz Analytics Toolkit (HeAT) - A Scientific Big Data Library for HPC -

Kai Krajsek
Forschungszentrum Jülich GmbH
Institute for Advanced Simulation
Jülich Supercomputing Centre (JSC)
52425 Jülich, Germany
k.krajsek@fz-juelich.de

Claudia Comito
Forschungszentrum Jülich GmbH
Institute for Advanced Simulation
Jülich Supercomputing Centre (JSC)
52425 Jülich, Germany
c.comito@fz-juelich.de

Markus Götz
Karlsruhe Institute of Technology
Steinbuch Centre for Computing (SCC)
Scientific Data Management
76128 Karlsruhe, Germany
markus.goetz@kit.edu

Björn Hagemeier
Forschungszentrum Jülich GmbH
Institute for Advanced Simulation
Jülich Supercomputing Centre (JSC)
52425 Jülich, Germany
b.hagemeier@fz-juelich.de

Philipp Knechtges
German Aerospace Center
Simulation and Software Technology
High-Performance Computing
51147 Cologne, Germany
Philipp.Knechtges@dlr.de

Martin Siggel
German Aerospace Center
Simulation and Software Technology
High-Performance Computing
51147 Cologne, Germany
Martin.Siggel@dlr.de

***Abstract*—We present HeAT, a scientific big data library supporting transparent computation on HPC systems. HeAT builds on top of PyTorch, which already provides many required features like automatic differentiation, CPU and GPU support, linear algebra operations and basic MPI functionality as well as an imperative programming paradigm allowing fast prototyping essential in scientific research. These features are generalized to a distributed tensor with a NumPy-like interface allowing to port existing NumPy algorithms to HPC systems nearly effortlessly.**

***Index Terms*—Big Data Analytics, HPC, Machine Learning, Deep Learning, Data Mining**

I. Introduction

Scientific Big Data Analytics has become an important instrument for tackling scientific problems characterized by the greatest data and computational complexity. Scientific data, e.g. MRI images, satellite data, detectors or numerical simulations on high-performance computers, are growing exponentially in nearly all scientific fields [1]–[4] pushing storage, processing, and analysis of such data to its limits. Traditional techniques for handling scientific data need to be replaced by specific solutions taking structure, variability and size of todays data sets into account. This paper presents the Helmholtz Analytics Toolkit (HeAT), a scientific big data analytics library for HPC systems enabling scientists to take full advantage of parallel high-performance computing with minimal programming effort on their side.

The large progress in big data analytics in general and machine learning/deep learning (ML/DL) in particular, has been considerably spurred by well-designed open source libraries like Hadoop, Spark, Storm, Disco, scikit-learn, H2O.ai, Mahout, TensorFlow, PaddlePaddle, PyTorch, Caffe, Keras, MXNet, CNTK, BigDL, Theano, Neon, Chainer, DyNet, Dask and Intel DAAL, to mention some of them. Despite the large number of existing data analytics frameworks, a library taking the specific needs in scientific big data analytics under consideration is still missing. For instance, no pre-existing library operates on heterogeneous hardware like GPU/CPU systems while allowing transparent computation on distributed systems. Typical big data analytics frameworks like Spark are designed for distributed memory systems and consequently do not fully exploit the shared memory architecture as well as the network technology of HPC systems. ML/DL frameworks like Theano or Chainer focus on single node computations or, when providing mechanisms for distributed computation, as done by TensorFlow or PyTorch, they impose the details of the distributed computation to the programmer. Libraries designed for HPC like Dask and Intel DAAL do not provide any GPU support. In the following, we will describe the core concepts of HeAT in order to fill the gap of existing big data libraries, and demonstrate its usage on a k-means cluster algorithm.

II. Co-design development approach

The library is designed and will be implemented in close cooperation with domain scientists within a scientific project, the Helmholtz Analytics Framework[1]. Eight scientific use cases from five different scientific fields (see Figure 1), i.e. earth system modeling, structural biology, aeronautics and aerospace research, medical imaging and neuroscience, have been chosen to ensure consideration of actual challenges of the specific scientific aspects of big data analytics. The use cases are tackling current research questions in their respective

This work is supported by the Helmholtz Association Initiative and Networking Fund under project number ZT-I-0003.

[1] http://www.helmholtz-analytics.de/helmholtz_analytics/EN/Home/home_node.html

fields that come to their limits with traditional data analytics methods.

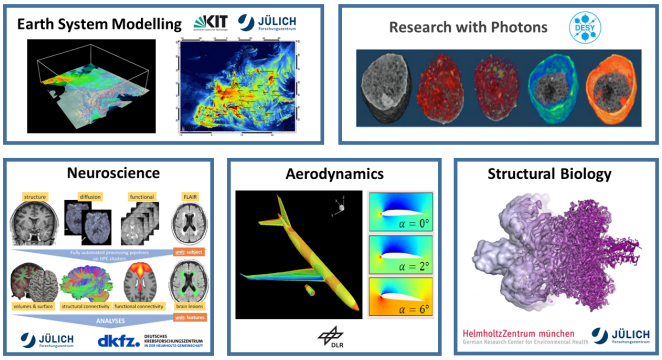

Fig. 1. Illustration of use cases from five scientific fields.

The techniques applied in the various use cases span over 20 different methods ranging from relatively light weight machine learning methods like k-means, or mean shift clustering, over frequent item set mining methods up to deep learning methods like convolutional neural networks for regression and classification tasks.

III. HeAT architecture

HeAT is based on a tensor data object on which basic scalar functions, linear algebra algorithms, slicing or broadcasting operations necessary for most data analytics algorithms can be performed. The tensor data objects reside either on the CPU or on the GPU and, if needed, are distributed over various nodes. Operations on tensor objects are transparent to the user, i.e. they remain the same irrespective of whether the tensor object resides on a single node or it is distributed over several nodes, allowing to conveniently port algorithms from single nodes to multiple nodes or from CPUs to GPUs. HeAT builds on top of PyTorch [5]. Development started in May 2018 and is, at the time of writing this paper, in an early pre-alpha phase. It is developed in the open, hosted on GitHub[2] and distributed under the MIT license. The basic design has been worked out and basic implementations have been carried out. A role model for

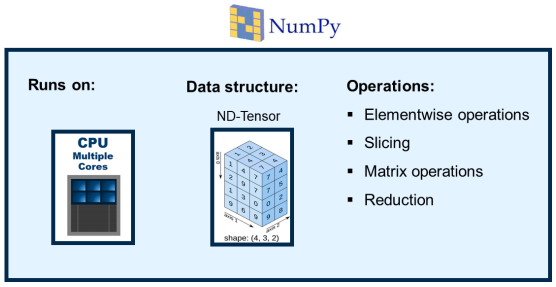

Fig. 2. The basic structure of the NumPy library: a tensor data structure and operations on top. The operations run transparently on multiple cores of one CPU.

HeAT is NumPy [6], a popular scientific Python library widely used for data analytics (see Figure 2). NumPy transparently makes use of all available CPU cores on one processor such that the user can focus on the algorithmic development without struggling with parallel programming issues. But NumPy has no further parallel programming features nor any GPU capabilities. In order to account for GPU computing and automatic differentiation we decided to rely on a modern tensor library. Overall, we examined 16 deep learning and big data libraries with respect to their properties and selected four of them for a benchmark with respect to memory consumption, CPU as well as GPU runtime: PyTorch, MXNet [7], TensorFlow [8] and ArrayFire [9]. As a result of the benchmark, we chose PyTorch as the backend for our HeAT library. Detailed results of the benchmark will be published separately. PyTorch is a

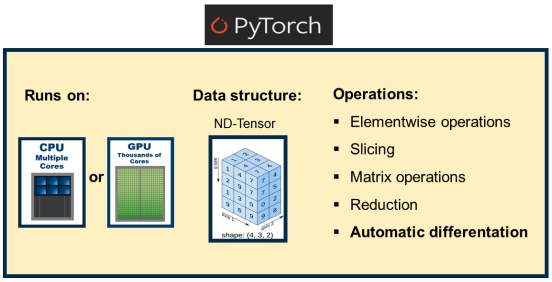

Fig. 3. The basic structure of the PyTorch library: A tensor data structure and operations as well as automatic differentation on top. The operations run transparently on multiple cores of one CPU or on one GPU.

deep learning library originally developed for neural network training and inference (see Figure 3). Its core module can be considered as an extension to NumPy with respect to automatic differentiation and GPU computation. It supports a subset of the NumPy operations and provides own operations required for artificial neural networks. A PyTorch tensor can be labeled to be differentiable and all subsequent operations are traced within a dynamical computational graph. The derivative of any transformed tensor with respect to the differentiable tensor can then be obtained with just one command due to the involved automatic differentiation mechanism. Computations on the GPU are automated, too. The PyTorch tensor is transferred onto the GPU by a single command or constructed directly on the GPU. PyTorch operation commands remain the same as for the CPU. When it comes to distributed computation, PyTorch supports several frameworks, i.e. TCP, GLOO, MPI and NCCL. However, details of the distribution of tensors on different nodes as well as the communication between the nodes need to be managed by the user.

HeAT builds upon PyTorch, providing an additional layer for distributed computation on GPUs as well as CPUs based on MPI (see Figure 4). Operations on tensor objects are transparent to the user, i.e. they remain the same irrespective of whether the tensor object resides on a single node or it is distributed over several nodes, allowing to conveniently port algorithms from single nodes to multiple nodes or from CPUs to GPUs. The basis of HeAT is a tensor object, an ND

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

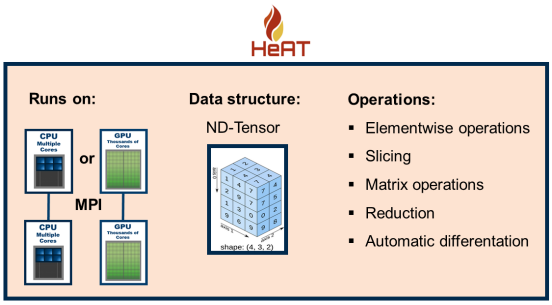

Fig. 4. The basic structure of the HeAT library: A tensor data structure and operations as well as automatic differentation on top. The operations run transparently on multiple cores of multiple CPUs or on mulptiple GPUs.

array structure of homogeneous numerical values. The tensor object is, if requested, split into several subsets along one selected dimension, whereby each subset belongs to one MPI rank (see Figure 5). The tensor object is directly created on different MPI ranks and filled with predefined values, e.g. equal values or random numbers. Alternatively, values are loaded from disc by parallel I/O via parallel HDF5 or parallel NetCDF. Operations on the HeAT tensor object can then be applied transparently, i.e. the user does not need take care about data transfer between the MPI ranks. The design of the HeAT operations follows the NumPy convention as far as possible, i.e. in the ideal case an algorithm implemented in NumPy can be ported to HeAT by simply exchanging NumPy operations with their HeAT counterparts. To this end, NumPy functions and methods are re-implemented using PyTorch and MPI4Py [10]. As an example, consider the creation of a one dimensional tensor filled with evenly spaced float values within a given interval running on three MPI ranks:

```
import heat as ht
range_data = ht.arange(6, split = 0)
```

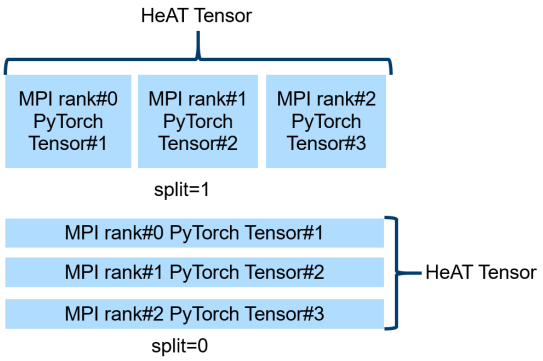

Fig. 5. Illustration of the splitting mechanism of the HeAT library on a two dimensional tensor. The tensor is equally distributed among the three requested MPI ranks. The HeAT tensor subset and each rank is realized by a PyTorch tensor. Splitting is supported in one of the two dimensions.

After importing the HeAT module, a tensor containing the numbers from 0. up to 5. is created. Internally, a subset containing values 0. and 1. is attached to rank number zero, the values 2. and 3. are attached to rank number one and the last two numbers are attached to rank number three. Subsequent operations can then be applied to the tensor object without caring about its distributed nature. For instance, the maximum of the tensor object can be obtained by the `argmax` method:

```
range_data.argmax()
>>>5
```

Also, computing the sum over all elements correspond to its NumPy counterpart:

```
range_data.sum()
>>>15
```

In order to support deep learning approaches and other ML methods requiring gradient based optimization, the automatic differentiation mechanism proposed in [11] will be extended to distributed computation. In a first step, a corresponding distributed adjoint operation is implemented for each HeAT tensor. Note that the PyTorch automatic differentiation mechanism can be re-used for all pointwise operations. If an operation is performed on a HeAT tensor being marked as differentiable, references to the operation, to its results as well as to the operation's arguments are stored in an object constituting a node in a dynamical computational graph. The references to the operation arguments are the edges to the parents of the dynamical graph. In order to perform back-propagation, we need the topological order of the graph. This order is obtained by storing a list tracking the order of the transformations applied to each differentiable tensor, i.e. we store a history of transformation for each differentiable tensor. In order to obtain the derivative of any node with respect to a differentiable tensor, the corresponding lists are traversed in reverse order. At each position in the list, the derivative of the output with respect to the input is computed using the corresponding stored node object.

IV. Example: K-means

As a demonstration of the library we describe how to port a k-means [12] NumPy implementation to its HeAT counterpart. We sketch the important steps of the algorithm by comparing NumPy and HeAT code snippets. The full HeAT k-means implementation can be found at https://github.com/helmholtzanalytics/heat/tree/master/heat/ml/cluster. K-means is a clustering algorithm that groups a set of data points with a predefined number of clusters according to the minimization problem

$$\arg\min_{\mathcal{C}} \sum_{i=1}^{k} \sum_{x \in \mathcal{C}_i} \|x - \mu_i\|^2 \tag{1}$$

where μ_i denotes centroid i, $\mathcal{C}_i$ denotes cluster i and k denotes the number of clusters. A local minimum of the optimization problem (1) can be obtained by the algorithm:

1) Choose k centroids
2) For each data point calculate the distance to all centroids
3) Assign each data point to the cluster with the closest centroid
4) Estimate new centroids as the mean of their corresponding cluster points
5) Go to 2 until convergence

Before one can apply the first step of the k-means algorithm, the data points to be clustered need to be loaded in the corresponding NumPy arrays as well as HeAT tensors. Whereas in the NumPy implementation the data are loaded into the NumPy arrays as a whole data block, in the HeAT implementation, if HeAT is running in distributed mode, only the data needed by the corresponding rank are loaded by the parallel I/O mechanism. All consecutive operations on the constructed arrays/tensors are equal or differ only with respect to small details. After choosing k initial centroids we need to compute the distance (step 2) of each point to the centroids and determine the index of the smallest distance. With NumPy, the second step can be realized by

```
distances = ((data - centroids) **
    2).sum(axis=1, keepdims=True)
matching_centroids =
    np.expand_dims(distances.argmin(axis=2),
    axis=2)
```

where `data` is a NumPy array of size $(n, m, 1)$ containing n m-dimensional data points and `centroids` is a NumPy array of size $(1, m, k)$ containing the initially chosen centroids. The corresponding HeAT implementation reads

```
distances = ((data - centroids) **
    2).sum(axis=1)
matching_centroids = distances.argmin(axis=2)
```

where the only differences stem from the fact that HeAT keeps dimensions after `sum` and `argmin` operations. Assigning the data points to their closest centroids (step 3) differs in NumPy

```
selection = (matching_centroids ==
    i).astype(np.int64)
```

from HeAT

```
selection = (matching_centroids ==
    i).astype(ht.int64)
```

by the build-in data types. The estimate of the new centroids (step 4) in NumPy is

```
new_centroids[:, :, i:i + 1] = ((data *
    selection).sum(axis=0. keepdims=True)
selection.sum(axis=0).clip(1.0, sys.maxsize))
```

and in HeAT

```
new_centroids[:, :, i:i + 1] = ((data *
    selection).sum(axis=0)
selection.sum(axis=0).clip(1.0, sys.maxsize))
```

The only difference is given by the way dimensions are kept after the `sum` operation.

V. Summary

We presented HeAT, a scientific big data library. After motivating the need for an additional big data analytics library in the scientific context, we described its core design principles, i.e. a distributed tensor object with transparent operations on top, as well as the design of the automatic differentiation mechanism. We finally illustrated the usage of the HeAT library by porting the k-means cluster algorithm from NumPy to HeAT demonstrating the close similarity of their user interfaces.

References

[1] R. N. Boubela, K. Kalcher, W. Huf, C. Nasel, and E. Moser, "Big data approaches for the analysis of large-scale fMRI data using Apache Spark and GPU processing: A demonstration on resting-state fMRI data from the human connectome project," *Frontiers in Neuroscience*, vol. 9, p. 492, 2015.

[2] F. Bamberg, H.-U. Kauczor, S. Weckbach, C. L. Schlett, M. Forsting, S. C. Ladd, K. H. Greiser, M.-A. Weber, J. Schulz-Menger, T. Niendorf, T. Pischon, S. Caspers, K. Amunts, K. Berger, R. Blow, N. Hosten, K. Hegenscheid, T. Krncke, J. Linseisen, M. Gnther, J. G. Hirsch, A. Khn, T. Hendel, H.-E. Wichmann, B. Schmidt, K.-H. Jckel, W. Hoffmann, R. Kaaks, M. F. Reiser, and H. a. Vlzke, "Whole-body MR imaging in the german national cohort: Rationale, design, and technical background," *Radiology*, vol. 277, no. 1, pp. 206–220, 2015. PMID: 25989618.

[3] J.-G. Lee and M. Kang, "Geospatial big data: Challenges and opportunities," *Big Data Research*, vol. 2, pp. 74–81, 2015.

[4] T. C. P.-G. Consortium, "Computational pan-genomics: Status, promises and challenges," *Briefings in Bioinformatics*, vol. 19, no. 1, pp. 118–135, 2018.

[5] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017.

[6] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed 15.11.2018].

[7] 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.," *CoRR*, vol. abs/1512.01274, 2015.

[8] 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.

[9] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, "ArrayFire - A high performance software library for parallel computing with an easy-to-use API," 2015.

[10] L. Dalcín, R. Paz, and M. Storti, "MPI for Python," *J. Parallel Distrib. Comput.*, vol. 65, pp. 1108–1115, Sept. 2005.

[11] D. Maclaurin, *Inference and Optimization with Composable Differentiable Procedures*. Dissertation, Harvard University, 2016.

[12] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, (Berkeley, Calif.), pp. 281–297, University of California Press, 1967.