Original research article

The International Journal of High Performance Computing Applications
2026, Vol. 0(0) 1–17
© The Author(s) 2026

Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420261444764
journals.sagepub.com/home/hpc

Large-scale coastal anomaly detection using the parallel machine learning library Heat

Wadim Koslow[1], Fabian Hoppe[1], Kathrin Rack[1], Hakan Akdag[1,2], Alexander Rüttgers[1] and Achim Basermann[1]

Abstract

Large-scale Earth observation (EO) datasets are crucial for identifying environmental changes, particularly in coastal regions vulnerable to erosion. However, analyzing these massive datasets requires computational techniques beyond the capabilities of conventional workstations or single nodes of an HPC system. In this study, we demonstrate how Heat, a highly parallel, open-source, Python-based library designed for scalable machine learning and data processing, can address this issue–quasi as an off-the-shelf solution for up-scaling–when detecting anomalies in satellite imagery of the German North Sea coast. Our implementation of an unsupervised outlier-based anomaly detection algorithm uses Heat's distributed arrays and vectorized map (vmap) primitives. This algorithm processes tens of millions of shoreline locations by sharing memory and computation across multi-node CPU/GPU clusters, with minimal code changes compared to NumPy/PyTorch. Weak-scaling experiments on both CPUs and GPUs demonstrate the scalability of our approach with increasing amounts of data. We thereby provide, to the best of our knowledge, the first density-based anomaly detection on large-scale EO datasets in a multi-node setting, with proven portability to different hardware architectures (x86-and ARM-based CPUs) and vendors (Nvidia and AMD GPUs). The resulting anomaly maps align with known severe weather episodes, daily anomaly counts correlate with wind metrics from coastal stations, and hotspot maps identify regions of high activity. These findings support the geophysical plausibility of the detections. Our approach is reproducible due to its deterministic algorithms, extensible to additional EO modalities (e.g. coherence), and broadly applicable to nationwide monitoring. Although the 20 m data resolution limits detection sensitivity, the method itself is resolution-agnostic.

Keywords

earth observation, anomaly detection, coastal erosion, machine learning, high-performance computing, distributed arrays

1. Introduction

In recent years, the availability and volume of Earth observation (EO) data have increased enormously, offering new opportunities to study environmental processes and detect significant changes on a large scale. For example, the *German Satellite Data Archive*[1] (D-SDA) houses over 20 petabytes (PB) of data and grows by approximately 3 PB annually due to new satellite missions. However, this significant growth in data volume poses substantial computational challenges. Many EO workflows rely — at least partially — on popular Python libraries, such as *NumPy*, *SciPy*, and *scikit-learn*[2]. While these libraries are effective for rapid prototyping and moderate data sizes, they face limitations on workstations due to their reliance on shared-memory parallelism. This restricts analyses to the memory capacity and computational power of a single machine. Additionally, these libraries usually lack support for GPU acceleration, which severely limits performance when analyzing large datasets.

In order to overcome these limitations, HPC infrastructures with massively parallel CPU and GPU clusters are essential. However, effectively exploiting these resources requires specialized software that combines ease of use, scalability, and high computational efficiency. The

[1]High-Performance Computing Department, Institute of Software Technology, German Aerospace Center(DLR), Cologne, Germany
[2]Faculty of Information Science and Communication Studies, Institute of Information Science, Cologne University of Applied Sciences (TH Köln), Cologne, Germany

Corresponding author:
Wadim Koslow, High-Performance Computing Department, Institute of Software Technology, German Aerospace Center (DLR), Cologne, Linder Höhe, 51147 Köln, Germany.
Email: wadim.koslow@dlr.de

Helmholtz Analytics Toolkit (Heat), a Python-based library developed by research institutions, including the German Aerospace Center (DLR), Forschungszentrum Jülich, and the Karlsruhe Institute of Technology, meets these requirements; see, e.g., Götz et al. (2020); Hoppe et al. (2025). Heat provides a scalable, distributed-memory, GPU-accelerated computational environment specifically tailored for large-scale scientific data analytics. Being a general-purpose, off-the-shelf solution targeting usage also by non-experts, Heat offers a fair compromise between performance and scalability on the one hand and ease of use and re-usability on the other hand. Currently, the most closely related libraries to Heat are Dask (Rocklin, 2015), which is widely used in the EO community, and Jax (Bradbury et al., 2018). For an overview on the Python ecosystem on distributed array computing and machine learning we refer to Hoppe et al. (2025), and for a review on high-performance Python for machine learning and data analytics in general to Castro et al. (2023). The related topic of parallelization strategies in the context of deep learning is addressed by Ben-Nun and Hoefler (2019).

This study uses Heat to identify coastal anomalies along the entire German North Sea coast by analyzing satellite imagery. We employ a massively parallel implementation of density-based anomaly algorithms, targeting regions with significant anomalous changes indicative of erosion. Although the underlying algorithmic ideas have been introduced and validated on synthetic anomalies in Koslow et al. (2026) for a small test region, up-scaling to larger regions and higher resolution images remains challenging. In this study, we use Heat to upscale the previously developed algorithms, enabling us to identify coastal anomalies along the entire German North Sea coast. We evaluate Heat's computational performance by extending the coastline window out to the Wadden Sea, creating a real-world weak scaling analysis. We compare CPU and GPU efficiencies and present results demonstrating strong alignment with documented severe weather events, underlining the practical relevance and effectiveness of our approach.

To clearly position our work within the landscape of large-scale EO solutions, we note that many density-based methods for anomaly detection use algorithmic approximation, grid-based decomposition, or an approximated nearest-neighbour search, cf. Thudumu et al. (2020); Corain et al. (2021); Okkels et al. (2025); Almansoori and Telek (2025). These approaches modify the original local outlier factor (LOF) algorithm, obtaining versions which facilitate scalable application to big data. Our approach pursues a similar overall concept; however by emphasizing localization, it significantly differs from the strategies reported in the literature.

The remainder of this article is organized as follows: The *German Sea Coast Dataset* section introduces the study area, the multi-year SAR stack, and the preprocessing steps. It also covers the weather data used to validate the detected anomalous events. Next, the *Heat software* section presents the architecture and design principles of the Heat library and the distributed array model. The *Methods* section then discusses the anomaly detection algorithms local outlier factor (LOF) and local outlier probabilities (LoOP). It also presents our novel, patch-based LOF/LoOP pipeline and its scalable implementation with Heat. The section on *Results and Discussion* reports on our numerical experiments, both in terms of performance metrics as well as anomaly detections and their correlations with major storm events. The final section summarizes the findings and discusses potential extensions.

2. German sea coast dataset

2.1. Dataset description

In DLR's project RESIKOAST ("Resilient supply infrastructure and goods flows in the context of coastal extreme weather events"), we study a wide range of the German North Sea coastline, from the island of Langeoog to the region north of Büsum. This approximately 2,000 km stretch exhibits diverse morphodynamics, ranging from macrotidal flats and estuaries in the west, which experience strong tidal effects, to microtidal sandy beaches and cliffs in the east, which experience low tidal effects. Consistently detecting changes in this heterogeneous littoral zone is essential for national adaptation planning in the face of rising sea levels and intensifying storms.

In this manuscript, we employ a seamless, multi-year stack of Sentinel-1 images acquired in interferometric wide swath (IW) mode that covers the entire German North Sea coast during the following time period: January 2016 - December 2023 with 13020 × 5680 pixels (≈363 acquisitions). The range of each image strip is approximately 250 km, yielding about 290 MB per scene in single-precision. Finally, all SAR products were geocoded to a position of 20 m to associate each pixel with its actual position on the ground using latitude/longitude coordinates. Additionally, we create coastline masks, which allows us to exclude irrelevant data from the mainland. An example can be seen in Figure 1. The method of generating these masks is described in detail in Koslow et al. (2026).

2.2. Weather data for validation

The validation by comparing our results to observed weather properties is based on measurements from the Deutsche Wetterdienst (DWD)[3] on coastal stations, which we obtain from the aggregator Meteostat[4]. We average the data of a chain of stations near the shoreline, including those in Langeoog, Spiekeroog, Wilhelmshaven, Bremerhaven, Cuxhaven, and Büsum. For each station, we retrieve the following data: average wind speed (wspd), wind peak gust (wpgt), barometric pressure, temperature (tavg, tmin, tmax),

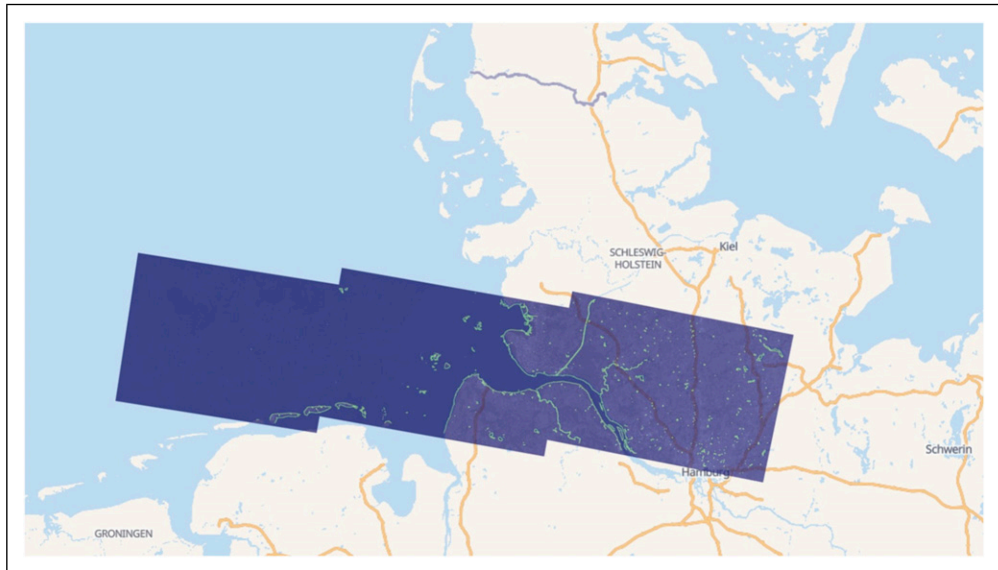

Figure 1. Illustration of the northern sea coast in the context of northern Europe. The green line inside the blue area depicts the coastline mask. Map data: © OpenStreetMap contributors, Open Database License (ODbL).

precipitation, air pressure (pres), and total sunshine duration (tsun).

Several severe storms within our observation window serve as natural experiments for anomaly validation, including Herwart (October 2017), Sabine (February 2020), Eugen (May 2021), and Malik/Nadia (January 2022).

3. Heat software - architecture and design principles

Heat[5] is a Python library for large-scale array processing, data analytics, and (classical) machine learning on HPC-systems, equipped both with CPUs and/or GPUs. Comprehensive technical details of Heat's internals are given in the original publication by Götz et al. (2015) and a more high-level introduction for a general research-software engineering audience has been provided by Hoppe et al. (2025). For the convenience of the reader we summarize the most important facts.

Heat is mostly based on PyTorch (Paszke et al., 2019) and MPI (Message Passing Interface Forum, 2023) via its Python interface mpi4py (Dalcín et al., 2008). This immediately ensures a high degree of interoperability and platform independence: in essence, PyTorch must be able to use the available hardware ("device") - which is the case, e.g., for a wide range of x86-and ARM-based CPUs, as well as for Nvidia- and AMD-GPUs - and the underlying MPI-implementation must be compatible with mpi4py and be able to deal with buffers on the respective devices (e.g., CUDA-aware MPI in the case of Nvidia-GPUs).

Heat's core abstraction is the distributed n-dimensional array (`heat.DNDarray`), which is a distributed-memory- and GPU-capable n-dimensional array that mimics NumPy's `ndarray` class. Conceptually, a `DNDarray` consists of "(process-)local arrays" that are PyTorch tensors on different MPI processes; hereby, the local arrays are the slices of the global array along a single axis/dimension (the "split axis"), a scheme of data distribution often referred to as slab-decomposition. Heat implements the classical Single Program Multiple Data (SPMD) model (Darema, 2001), i.e., every process executes identical Python code against different local arrays. The overall parallelization style can be described as hybrid-parallel and bulk synchronous as process-local computations using PyTorch's OpenMP/CUDA/ROCm-based tensor functionality alternate with communications between MPI processes. Moreover, eager execution is applied. This design is in contrast to, e.g., Dask (Rocklin, 2015), which is based on task-based parallelization and lazy execution.

Heat's API is simple and, whenever possible, close to the one of NumPy, SciPy, and/or scikit-learn in order to increase usability also for non-experts in HPC. In many cases, specifying the "split axis" illustrated in Figure 2 is the only contact the user needs to have with parallelization aspects. Roughly speaking, the libraries goal is to make array computing, data analytics and machine learning as simple on a supercomputer as it is on a workstation with NumPy/SciPy/scikit-learn.

4. Methods

In this section, we describe the following density-based anomaly detection approaches: the LOF published by

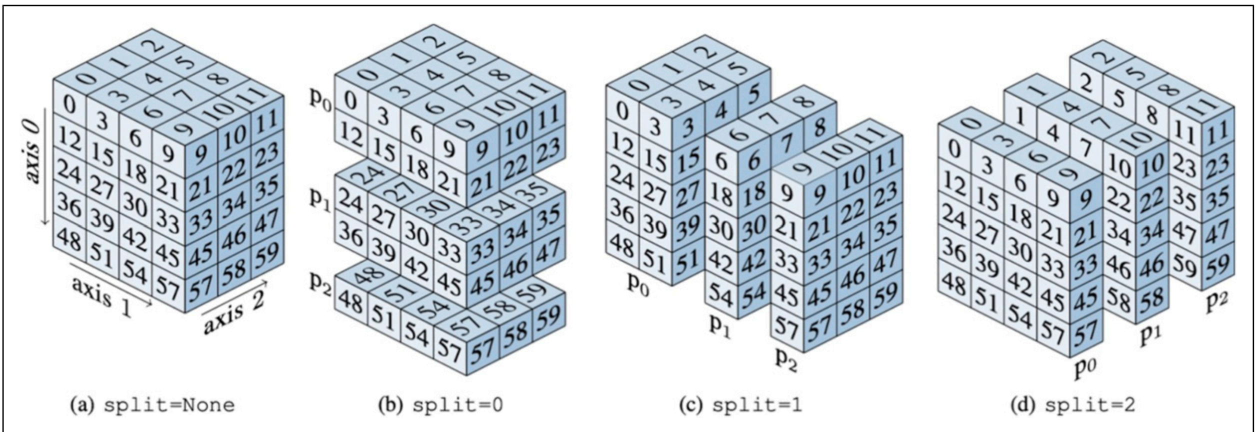

Figure 2. A distributed Heat tensor is composed of multiple local PyTorch tensors that can be split along an arbitrary axis. Reprinted with permission from Götz et al. (2020), © 2020 IEEE.

Breunig et al. (2000), its probabilistic variant local outlier probabilities (LoOP) described by Kriegel et al. (2009), and a novel patch-wise LOF-based method tailored to coastline pixels. LOF and LoOP identify deviations by comparing a sample's local neighborhood density to that of its neighbors (LoOP returning calibrated probabilities in [0,1]); our patch-wise LOF described in detail and validated on synthetic anomalies in Koslow et al. (2026) extends this idea by applying independent LOF models to the temporal trajectories of centered coastal patches, enabling pixel-scale anomaly flags and time aggregated hotspot maps. Density-based approaches such as LOF and LoOP have proven effective across diverse domains, including monitoring complex multi-modal industrial processes (Ma et al., 2013) and analyzing video from hybrid rocket combustion experiments (Rüttgers and Petrarolo, 2021). Density based methods require the determination of nearest neighbors and this step becomes increasingly expensive as the data set size grows. In a big-data setting, several parallelization strategies have been developed, as described inAdesh et al. (2024); Yan et al. (2017); Alshawabkeh et al. (2010). A classical HPC-oriented MPI-based parallelization of DBSCAN, a clustering algorithm closely related to LOF, was presented in Götz et al. (2015).

4.1. Local outlier factor

Given a set of samples $X = \{x_1, \ldots, x_T\} \subset \mathbb{R}^d$, a metric $d(\cdot,\cdot) : X \times X \rightarrow \mathbb{R}_{\geq 0}$ for measuring pairwise distances, and a hyperparameter $k \in \mathbb{N}$, we introduce the following concepts:

k-distance and -neighborhood: The k-distance of x is the distance to its k-th nearest neighbor according to

$$k - \text{distance}(x) \quad = \min \{\epsilon \in \mathbb{R}_{\geq 0} \ : \ \text{card } (\{y \neq x : d(x,y) \leq \epsilon\}) \geq k\} \qquad \text{for } y \in X \text{ and card } (\cdot) \text{ as cardinality.}$$

The (inclusive) k-neighborhood collects all points at most this far is defined as

$$N_k(x) = \{y \neq x \ : \ d(x,y) \leq k - \text{distance}(x)\}.$$

Reachability distance: To mitigate the effects of having very close neighbors, the LOF algorithm uses the reachability distance between x and y:

$$\text{reach} - \text{dist } (x,y) = \max\{k - \text{distance}(y), d(x,y)\}.$$

Local reachability density (LRD): The inverse of the average reachability distance from x to its neighbors defines the local density estimate:

$$\text{lrd}_k(x) = \frac{\text{card } (N_k(x))}{\sum_{y \in N_k(x)} \text{reach} - \text{dist } (x,y)}.$$

Local Outlier Factor: Finally, the LOF compares the local density around x to the densities of its neighbors:

$$\text{LOF}_k(x) = \frac{1}{\text{card } (N_k(x))} \sum_{y \in N_k(x)} \frac{\text{lrd}_k(y)}{\text{lrd}_k(x)}. \tag{1}$$

LOF determines the extent to which x is an outlier by comparing its local reachability distances to those of its neighbors. If $\text{LOF}_k(x)$ significantly exceeds 1, i.e., if the local reachability density of x is significantly lower than the one of its neighbors, it is identified as a potential outlier. Conversely, if $\text{LOF}_k(x) \leq 1$, then the observation is a potential inlier. The hyperparameter k governs the smoothness of the distance metric and needs to be chosen depending on the application. Since LOF scores tend to change smoothly with k, selecting the correct order of magnitude is usually sufficient. The section on the patch-wise LOF provides information on how k was chosen for our application as a patch-wise algorithm.

4.2. Local outlier probabilities

LoOP converts the density contrast idea of LOF into a calibrated probability in [0,1]. It relies on a probabilistic distance built from the dispersion of distances in a local neighborhood and a significance parameter λ. With the same notation as in the section on the LOF, the LoOP score is obtained as follows:

Standard and probabilistic distance: Standard distance, also known as local dispersion, is defined as

$$\sigma(x) = \left(\frac{1}{\text{card } (N_k(x))} \cdot \sum_{y \in N_k(x)} d(x,y)^2 \right)^{1/2},$$

whereas probabilistic distance is introduced as

$$\text{pdist } (\lambda, x) = \lambda \cdot \sigma(x) \text{ with } \lambda > 0;$$

with a hyperparameter $\lambda \in \mathbb{R}_{>0}$; see the paragraph below for its meaning.

Probabilistic Local Outlier Factor and neighborhood-based normalizer: The probabilistic local outlier factor (PLOF) compares the local dispersion around x to that of its neighbors and is computed as follows:

$$\text{PLOF }_\lambda(x) = \frac{\text{pdist } (\lambda, x)}{\frac{1}{\text{card } (N_k(x))} \sum_{y \in N_k(x)} (\text{pdist } (\lambda, y))} - 1.$$

Here, if $\text{PLOF}_\lambda(x) > 0$, i.e., if x is more dispersed than its neighbors, this indicates a potential outlier. Conversely, if $\text{PLOF}_\lambda(x) \leq 0$, then the local dispersion is similar to neighbors and x is a potential inlier. With this, the neighborhood-based normalizer is defined by

$$\text{nPLOF} := \lambda \cdot \sqrt{\mathbb{E}\left[(\text{PLOF }_\lambda)^2\right]},$$

where $\mathbb{E}[\cdot]$ denotes the expectation with respect to the empirical uniform distribution over the set of k-nearest neighbors or, in other words, the arithmetic mean. Therefore, it can be seen as the standard deviation of PLOF, assuming a mean of 0.

Local outlier probability LoOP: The final LoOP score is then given by

$$\text{LoOP } (x) := \max\left\{0, \text{erf } \left(\frac{\text{PLOF }_\lambda(x)}{\text{nPLOF } \cdot \sqrt{2}} \right) \right\}, \quad (2)$$

where erf $(\cdot)$ is the Gaussian error function that maps the score to [0, 1].

Due to construction it holds LoOP $(x) \in [0, 1]$ for every $x \in X$. Hence, these values can be understood as the probability that the corresponding data point is an outlier. As the LoOP method yields *normalized* scores, it has the advantage (compared to LOF) that its results for different datasets can be compared directly. The hyperparameter λ is a scaling factor that controls how strongly distance variability influences the outlier score and can be understood as a confidence multiplier.

4.3. Patch-wise LOF-based anomaly detection

For coastal risk assessment, spatially-resolved anomaly maps along the shoreline must be provided. Therefore, applying LOF directly to the time series of entire images is not suitable, as this could only identify the *whole image* (out of 363) as anomalous with respect to the others. In order to detect local anomalies in space and time, we present a patch-wise LOF-based method that operates on short, spatially anchored neighborhoods. We provide a brief summary of the main idea in the following and refer interested readers to Koslow et al. (2026) for a detailed exposition, a comparison with different deep neural network approaches such as autoencoders and a validation on synthetic anomalies. We also note that the algorithm described can be similarly applied to LoOP.

Let the geocoded raster domain be $\Omega \subset \mathbb{Z}^2$ and the shoreline mask be $\mathcal{M} \subset \Omega$ (cf. Section *Dataset description*). For each date $t \in \{1, \ldots, T\}$ and for each of the $N =$ card $(\mathcal{M})$ shoreline pixels $i \in \mathcal{M}$, extract a centered $p \times p$ window (default $p = 7$) and vectorize it according to

$$\boldsymbol{x}_{i,t} = I_t[\mathcal{N}_p(i)] \in \mathbb{R}^{p^2},$$

where $I_t\,[\cdot]$ is the SAR image at date t restricted to the indicated pixel subset, and $\mathcal{N}_p(i)$ is the $p \times p$ neighborhood around i, where i is at least p pixels away from the edge so that this neighborhood is able to exist. For each i we obtain a per-pixel time series

$$D_i = \{\boldsymbol{x}_{i,1}, \ldots, \boldsymbol{x}_{i,T}\} \text{ with } i \in \mathcal{M} \quad (3)$$

as illustrated in Figure 3.

Using the LOF definitions of the LOF section with the Euclidean distance and neighborhood size k, we compute an LOF score $s_{i,t}^{(k)}$ for each element $\boldsymbol{x}_{i,t}$ in the pixel-specific set D_i as visualized in Figure 3. To stabilize the detections, we sweep k over a small set of values $K = \{20, 21, \ldots, 60\}$ and aggregate via

$$S_{i,t} = \max_{k \in K} \left(s_{i,t}^{(k)} \right) \quad (4)$$

but other robust aggregations such as trimmed mean are also possible.

To translate these anomaly scores into a binary decision on whether a pixel is as an outlier or not, we introduce an anomaly threshold τ, e.g., a constant such as 1.5 (LOF default in *scikit-learn*) or the 99.7th percentile, which corresponds to approximately three standard deviations under the assumption of a Gaussian distribution. A pixel $i \in \mathcal{M}$ is classified as anomalous at time t if and only if $s_{i,t} > \tau$. The binary classification vector on all dates of pixel i is denoted by C_i. This allows us to directly generate anomaly maps for each date, cf. Figure 3. Alternatively, for ranking instead of hard decisions, the scores $s_{i,t}$ could also be visualized after per-date min–max scaling.

In practice, identifying high-risk regions, such as areas prone to coastal erosion, is crucial. For this

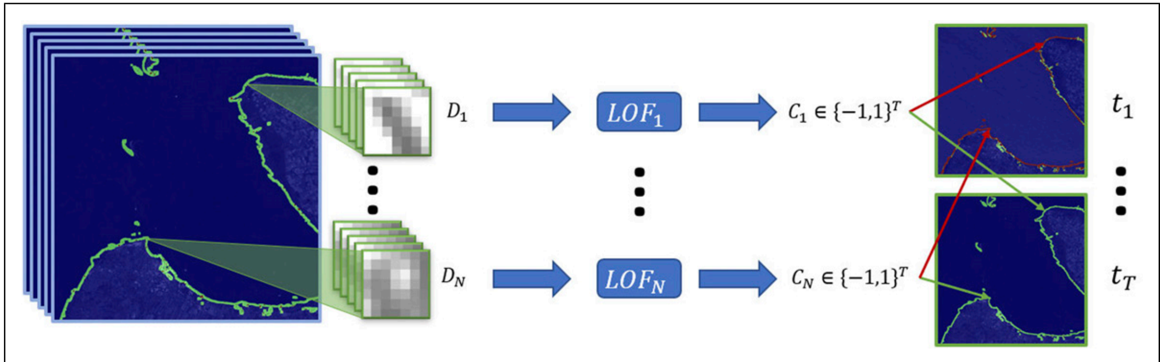

Figure 3. Illustration of the patch-wise workflow: For each subset $D_{i,\ i\in\{1,\dots,N\}}$, the LOF algorithm is computed independently. After thresholding, each local time series results in a binary classification vector C_i with length T and elements in $\{-1, 1\}$ (here: inlier = -1 (green), outlier = 1 (red)). Aggregating individual $\{C_i\}$ values across all shoreline pixels produces date-wise anomaly maps. The right side shows examples of our results in the Cuxhaven region.

purpose, the frequency of detection in an area is most important. The hotspot map provides this information and is defined as temporal average of the anomaly regions according to

$$H(i) = \frac{1}{T}\sum_{t=1}^{T} \mathbb{1}_{\{S_{i,t} > \tau\}}, \quad i \in \mathcal{M}; \tag{5}$$

herein, $\mathbb{1}$ denotes the element-wise indicator function. This hotspot map highlights shoreline sectors frequently flagged as anomalous and guides follow-up inspection.

4.4. Scalable anomaly detection with Heat

Obviously, computing the localized LOF scores is the computationally most demanding part of our anomaly detection pipeline: for every pixel on the coastline, $i \in \mathcal{M}$, LOF scores of the corresponding time-series of $p \times p$-neighborhoods, i.e., a data set consisting of T samples of dimension $d = p^2$ (after flattening the neighborhoods), need to be computed We expect high scalability, since LOF can be applied independently to each of these pixel-wise time series and the costly computation of nearest neighbors itself does not need to be scaled up to the memory-distributed setting. However, creating an array of the localized time series data of shape $N \times T \times p^2$ from the original images is highly memory-intensive, since the number of coastline pixels, N, is quite high.

The following step, computing LOF scores is even more challenging. A naive loop over all N pixels is prohibitive due to runtime constraints. Furthermore, parallelization must take into account the additional memory overhead due to the concurrent computation of several distance matrices of shape $T \times T$ (which might be larger than $T \times p^2$). In the following we describe how our initial, purely PyTorch-based, prototype of the pipeline, capable of being run on a subset of the entire coastline, has been scaled up to the distributed-memory setting required for dealing with the entire German coastline.

Initially, the T images of shape $H \times W$ are loaded into a `DNDarray` of shape `(T, H, W)` split along the first axis ("`split=0`") such that time is shared across the ranks. This allows to extract the $p \times p$-patches centered at pixels belonging to the coastline by applying a certain number of subsequent (PyTorch-)convolutions to the non-split dimensions; a `DNDarray` of shape `(N,T,p,p)` with `split=1` is obtained. Heat's `resplit_()`-function allows to redistribute the data in order to have `split=0`, i.e., parallelization over the patches, with minimal memory overhead[6]. Finally, the patch dimensions are flattened, resulting in a `DNDarray` X of shape `(N,T,p ** 2)`, still with `split=0`.

Next, the actual computation of LOF scores is scaled up. Starting point is a pure PyTorch-implementation of LOF as described in the LOF section:

```
def lof_kernel(X: torch.Tensor, k: int=10):
    """
    X: the input data set, a torch.Tensor of shape (n_samples, d)
    k: number of neighbors, integer
    """
    ...
    return lofs
```

Given an input tensor of shape `(n_samples, d)`, it returns the LOF-scores `lofs` as tensor of shape `(n_samples,)`) The number of neighbors to be used in the algorithm, k is passed as a keyword argument. In order to apply `lof_kernel` in vectorized (w.r.t. Axis `0`) fashion to our `DNDarray` of shape `(N,T,p ** 2)`, we make use of Heat's `vmap` function:

```
vmapped_lof = ht.vmap(lof_kernel, chunk_size=cs)   # map over N
```

Yields a callable, that takes a `DNDarray` of shape `(B,n_samples,d)`, possibly split over multiple CPUs or GPUs along axis `0` ("`split=0`"), computes the LOF scores of each of the `n_samples` many `d`-dimensional data, and returns them as `DNDarray` of shape `(B,n_samples)`, again with `split=0`. Hereby, the vmapped function inherits the keyword arguments from the original function. The argument `chunk_size` allows to prescribe how many instances of `lof_kernel` can be applied simultaneously *per* MPI process; for additional technical details we refer to the Synthetic experiments on varying hardware section at the very end of the paper.

```
# X.shape = (N, T, p ** 2), X.split = 0; S.shape = (N, T), S.split = 0
S = vmapped_lof(X)
# assuming X is Gaussian, a typical threshold is the 3-sigma interval
mu     = S.mean(axis=0)
sigma  = S.std(axis=0)
thresh = mu + 3 * sigma
# create (boolean) anomaly map, A.shape = (T, H, W), A.split = 0
A = ht.ones(T,H,W)*-1
A[:, rwos, cols] = S > thresh
# H is hotspot map, H.shape = (H,W), not split anymore
H = A.sum(axis=0) / T
```

Once the LOF scores have been computed, statistics of these values and the resulting anomaly and hotspot maps, respectively, can be calculated using Heat's NumPy-like syntax:

This example demonstrates that most operations in Heat handle parallelization-related issues automatically and the syntax is–whenever possible–close to the one of NumPy, PyTorch etc., adaptation of an existing prototype workflow to a scalable setting is largely unproblematic. We summarize the shapes and parallelization configurations of the different data representations that occur in the anomaly detection pipeline in Table 1.

Expected runtime and memory consumption Let us briefly summarize the influence of the chosen parallelization approach on resource utilization. The underlying dataset has shape `(N, T, p ** 2)` and thus its memory consumption is already of order $\sim NTp^2$. During the computation of the LOF scores for a data set of shape $T \times p^2$, the pairwise distances must be calculated and stored, resulting in an additional memory usage of order $\sim T^2$. Hence, we expect the overall memory consumption of our workflow to be of order $NT(p^2 + T)$; it should be noted that in our case already the pure underlying data might exceed the memory available on a single machine. Parallelizing over r MPI processes reduces the size of data *per* process to $r^{-1}Np^2$. Applying vmapped LOF with `chunk_size=`s, therefore results in an overall memory consumption of order $\sim r^{-1}Np^2 + sT^2$ per process as s pairwise distances matrices are set up in parallel. If $\mathbb{T}_{\mathrm{LOF}}(s, T, p^2)$ denotes the runtime of LOF score computation for s instances of the $T \times p^2$ data set, the overall runtime is then expected to be of order $s^{-1}r^{-1}N \cdot \mathbb{T}_{\mathrm{LOF}}(s, T, p^2)$. Consequently, the chosen approach allows to adapt the memory consumption to the available hardware by choosing an appropriate number of processes and suitable chunk size for vmap.

5. Results and discussion

5.1. Scalability analysis for a real-world scenario

For a suitable scalability analysis, we prioritize weak-scaling experiments over strong ones. This is due to the fact that weak scaling reflects the computational reality of EO more realistically. In the application of anomaly detection to coastal regions, the challenge is typically not to accelerate a task of fixed complexity, but to handle ever-increasing data volumes, which are, for instance, driven by higher spatial resolutions and growing monitoring areas/time series that are investigated.

We conduct our weak scaling study, by adjusting the width of the coastal strip in such a way that the resulting

Table 1. Overview of data representations and their corresponding parallelization strategies used in the anomaly detection pipeline. The shapes consist of combinations of image height (H), image width (W), width of quadratic pixel patch (p), number of time steps (T), and number of pixels along the coastline (N).

Data representation	Shape	Parallelization
Single input image	$H \times W$	No split
Pixel patch	$p \times p$	No split
Full data	$T \times H \times W$	Split 0 (along time dimension)
Full data patched along coastline	$N \times T \times p^2$	Split 0 (along coastline pixels)
LOF scores	$N \times T$	Split 0 (along coastline pixels)
Anomaly map	$T \times H \times W$	Split 0 (along time dimension)
Hotspot map	$H \times W$	No split

number of N pixels, is roughly proportional to the number of computational resources. Expanding this strip is of interest because certain coastal change processes may initiate further offshore in the Wadden Sea before reaching the coast. The exact values can be seen in Table 2. All experiments were conducted on the CPU partition of DLR's HPC-system *terrabyte*[7], which is equipped with 2 Intel Xeon Platinum 8380 40C 270 W 2.3 GHz and 1024 GB RAM per node. We run 8 MPI processes per node and 20 threads per MPI process[8]. Runtime and memory consumption are measured using the Python library perun Gutiérrez Hermosillo Muriedas et al. (2023) which reads out the available hardware counters (psutil, Nvidia NVML etc.) and aggregates these results[9]. OpenMPI 4.1.5 and Python 3.10.10 are loaded from the module system of the cluster, whereas PyTorch 2.3.1+cu118, mpi4py 4.0.0, perun 0.9.0, and the current development version of Heat 1.6.0. dev0 are installed with pip in a Python virtual environment.

Figure 4 shows the runtime and the maximum memory usage of the weak scaling experiments. Additionally Table 3 shows the total runtimes of the programs. The main algorithms from the patch-wise LOF which are represented by "vmapped predict" exhibit near perfect weak scaling behavior. During pre- and post-processing, some operations are performed on full images of the North Sea that are independent of the chosen width of the coastline which is used to scale the size of the data. However, these operations also had to be parallelized with Heat, thus the presented results represent a mix of weak and strong scaling, which is also reflected in Table 3. Since the steps before and after the vmapped method are the same, the runtime and memory usage of these parts are nearly identical for both the LOF and LoOP. However, the actual algorithms do have significantly different runtimes, with the LoOP being faster then the LOF. This is especially interesting, as the situation is reversed, when both algorithms are run without vmap. This might indicate that some operations are better suited to be vectorized then others and that the LoOP has more favorable conditions compared to the LOF.

In addition to the weak scaling experiments on the CPU, we demonstrate the weak scalability on a GPU using a smaller initial dataset. The exact data sizes are again shown in Table 2. The experiments were performed on the GPU-partition of *terrabyte*, equipped with 2 Intel Xeon Gold 6336Y 24 C 185 W 2.4 GHz CPUs and 4 Nvidia HGX A100 80 GB 500 W GPUs per node; CUDA 11.8 is loaded from the module system. We compare the results to the CPU performance by running the same experiments on the CPU-partition, now with 4 MPI processes and 40 threads per node. This change was implemented to improve comparability with runs on GPU, for which 4 MPI ranks per node is the natural configuration. The scaling results are shown in Figure 5. Similar to the CPU performance, the experiments on GPU demonstrate the expected close-to-optimal weak-scaling capabilities while decreasing runtime compared to CPU,

Table 2. Data sizes for weak scaling on CPU and GPU nodes; "CL" stands for coastline. Scaling refers to the ratio of the number of pixels in one line to the previous line.

	On CPU			On GPU		
# Nodes	CL width	# Pixels	Scaling	CL width	# Pixels	Scaling
1	13	504.501	NA	2	128.797	NA
2	31	1.012.474	2.006	5	252.427	1.959
4	71	2.027.481	2.002	13	504.501	1.998
8	152	4.055.690	2.0003	31	1.012.474	2.006
16	420	8.120.538	2.002			

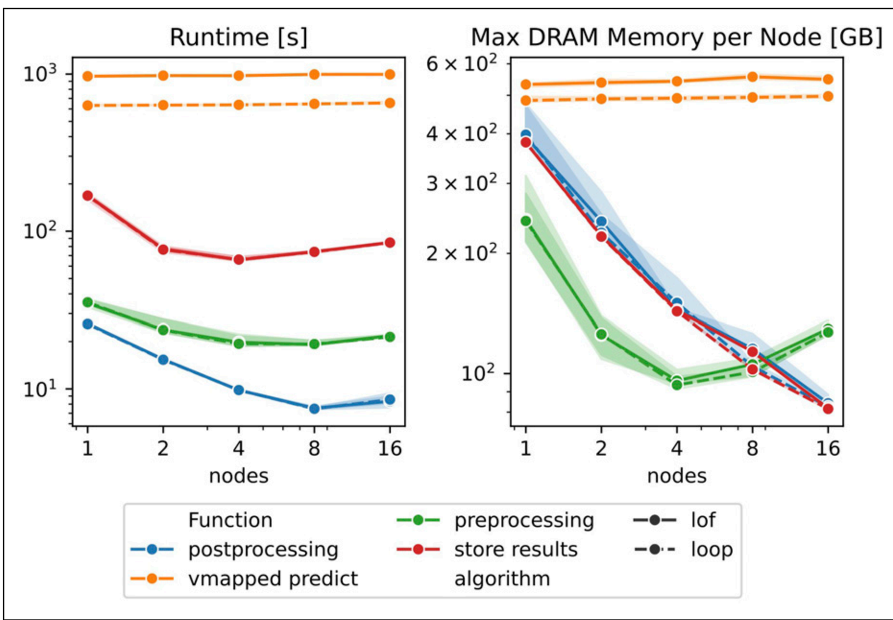

Figure 4. Weak scaling results on CPU. The left image shows the runtime of different functions in our workflow. Since "vmapped predict" contains the main LOF/LoOP calculations it dominates the entire runtime. The right image shows the maximum memory usage per node which is also dominated by the "vmapped predict" routine. Shaded areas and the lines indicate minimum/maximum and average over 10 runs, respectively.

however, with an unexpectedly low speedup. The latter might not be enough to justify the usage of the more expensive (in terms of hardware and power consumption) GPUs in practice, in particular as our pipeline is memory-intensive which makes CPUs with their larger DRAM a more natural choice than GPUs with usually smaller High Bandwidth Memory (HBM).

Furthermore, it is of interest to analyze the influence of the chunk size parameter, particularly with respect to memory usage. Therefore, we conducted additional experiments with varying chunk sizes, which are shown in Figure 5. For the considered range of chunk sizes and problem sizes, the chunk size does not have significant impact on the runtime. For CPUs, the maximum DRAM memory per node shows the expected behavior: the smaller the chunk size, the smaller the necessary maximum DRAM. On GPUs, we could not confirm this expectation on the chosen setting. One possible explanation for the intransparent behavior is the differing underlying implementation and resulting effects of vmap in PyTorch on CPUs and GPUs, in particular w.r.t. Caching mechanisms in the memory allocation.

Both the small speedup from CPUs to GPUs and the behavior w.r.t. Different chunk sizes illustrate the typical downsides of generic off-the-shelf approaches compared to highly problem-specific implementations customized to the hardware. While we will discuss some speculative partial explanations for these observations later on in the context of synthetic experiments on varying hardware, certainly further investigation–going beyond the scope of the present paper–would be necessary to determine the exact cause.

Table 3. Total runtime (seconds) across varying node counts, evaluating the scalability of the whole pipeline on CPU (Large/Small Coastline) and GPU architectures.

	CPU large coastline		CPU small coastline		GPU	
Nodes	**Lof**	**Loop**	**Lof**	**Loop**	**Lof**	**Loop**
1	1351.0	1012.7	877.5	732.6	726.1	665.8
2	1178.5	833.3	619.2	476.7	402.9	392.2
4	1129.4	787.1	554.7	453.2	357.4	231.8
8	1137.8	789.1	553.5	399.3	282.8	225.1
16	1147.0	805.3				

5.2. Coastal anomaly and hotspot maps

We assess the capabilities of our anomaly detection method, when applied to the whole north sea coast at once, for different scenarios as image artifacts and storm events. On smaller scale we already have shown, that the patch-wise approach works well for synthetic anomalies (Koslow et al., 2026). In the following, we preset the result for one scenario just for a single coastline width, because the outcomes for the various widths are similar.

Figure 6 shows the anomaly (top) and the outlier score map (bottom) generated with the LoOP algorithm for a high coastline thickness of 420 px. In the anomaly map, anomalies are marked in red, other examined pixels are marked in green and blue pixels are not considered. In the outlier score map, blue represents the

minimum value and red represents the maximum value. Both images accurately detect the artifact caused by radio frequency interference during the acquisition of the SAR image. The outlier score map even pictures the inner structure of the artifact, which has a more prominent deviation in the middle and fades outward.

Another real-world application is the detection of the impact of storm events. Figure 7 depicts the anomaly map of the North Sea shoreline with two underlying corresponding SAR images before a storm event on 2017-10-23 and during the storm Herwart on 2017-10-29, which raged over northern and eastern Germany with gale-force winds on October 2017-10-28 and 2017-10-29 as reported by the DWD[10]. Both images were generated with the LoOP algorithm and a coastline width of 13 px. In contrast to 2017-10-23, where the majority of pixels are marked green, on 2017-10-29, the majority is marked red, thus indicating anomalies as expected. Furthermore, the other mentioned storm events such as Sabine, Eugen and Malik are detected in the same manner.

In order to get a more general overview of the correlation between detected anomalies and daily weather conditions, we estimate the anomaly count per day and compare it to the average wind speed (*wspd*) on that day. Figure 8 shows that for both algorithms the number of anomalies (blue bars) is mostly associated with a high *wspd* (orange dots). In particular, we find the largest number of anomalies for the three storm events shown in the presented time range (marked by red crosses). For this comparison, the LOF algorithm seems to be slightly better than the LoOP algorithm, as LoOP detects a higher number of anomalies for days with low *wspd*.

In order to further generalize our investigations, we calculate the Pearson correlation coefficient (PCC) between the number of anomalies and the available weather properties. Figure 9 shows that there is a strong correlation between the daily anomaly count and the wind speed (*wspd*) as well as the wind peak gust (*wpgt*) for both the LOF and the LoOP, since $|PCC| > 0.5$ for both. For other weather properties there is only a minor correlation with $|PCC| < 0.3$.

Detecting specific dates with high anomaly counts, as presented above, is important to identify sudden changes in a huge dataset. However, it is also of interest to spot slight but continuous changes or recurring abnormalities, which would result in frequent appearance of anomalies. Therefore, we average the anomaly counts for every patch over time and generate a hotspot map, as depicted in Figure 10.

The image highlights two areas that exhibit a particularly high number of abnormalities during the period under consideration. On the left, the east Frisian island of Langeoog is shown to have a much higher number of abnormalities than its neighboring islands. This could be explained by increased storm activity during the winter of 2021 and 2022 which resulted in the initiation of a beach nourishment project. The project added 450,000 cubic meters of sand to restore the protective dunes and maintain the island's defenses against storm surges[11]. On the right side, the Cuxhaven harbor is shown. The higher number of anomalies in this region is most likely the result of increased activity in the area, e.g. from ships.

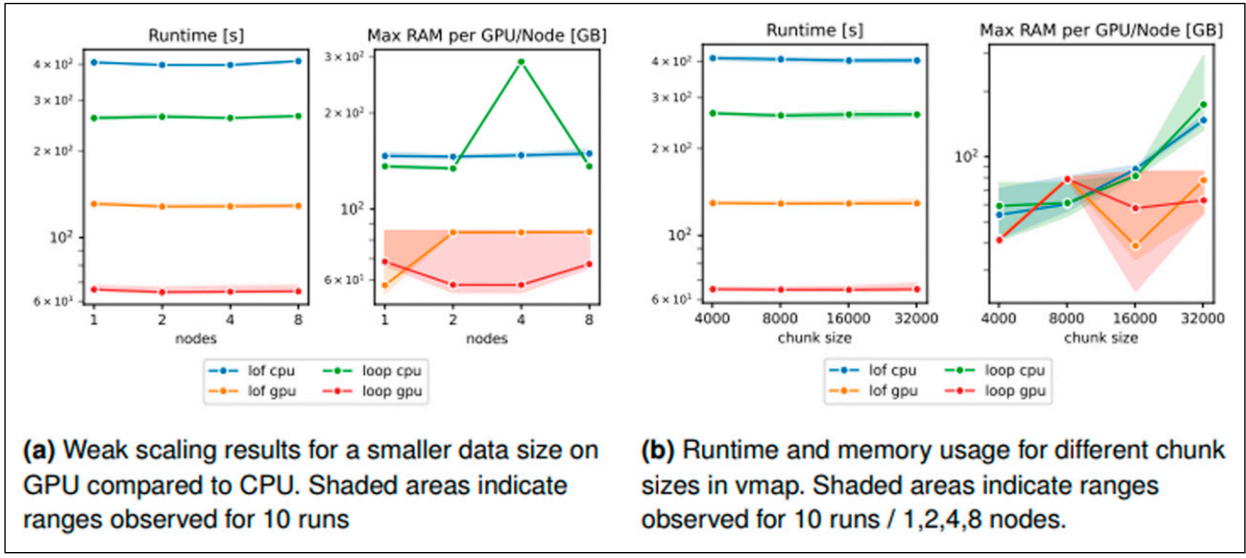

Figure 5. Comparison of GPU weak scaling results. (a) Weak scaling results for a smaller data size on GPU compared to CPU. Shaded areas indicate ranges observed for 10 runs (b) Runtime and memory usage for different chunk sizes in vmap. Shaded areas indicate ranges observed for 10 runs/1,2,4,8 nodes.

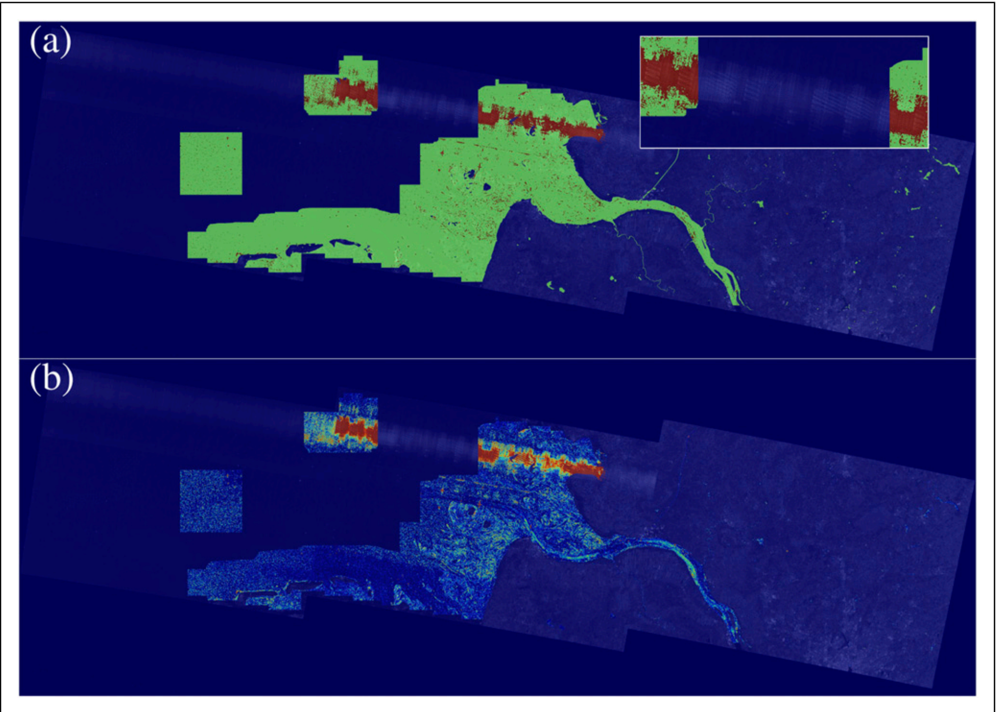

Figure 6. Illustration of an anomaly map (a) and an outlier score map (b) with an underlying SAR image, both generated from a LoOP analysis with a coastline width of 420 px on 2019-10-31. A visible artifact caused by radio frequency interference was detected in the upper part of both figures.

5.3. Synthetic experiments on varying hardware

In order to assess the influence of chunk size and the underlying hardware in more detail in a controlled environment, we conducted experiments using slightly smaller synthetic data. For p MPI processes, we selected a random dataset of shape $(50000 \cdot p, 300, 20)$, i.e., $50000 \cdot p$ time series, each with 300 time steps and 20 features, and computed LOF and LoOP scores for the number of neighbors k in the range 10-30, $\lambda = 3$ (LoOP significance parameter) and thresholds $\tau = 1.5$ (LOF) and $\tau = 0.9$ (LoOP), respectively. We considered the following hardware configurations with varying combinations of chunk size and number of MPI processes:

- "**Nvidia A100**": Experiments on the GPU-partition of the terrabyte cluster as above.
- **"Intel Xeon CPU":** Experiments on the CPU-partition of terrabyte with 8 MPI-processes per node as above.
- **"AMD MI250":** Experiments on MI250-nodes of the NHR@KIT future technologies (FTP) partition: the two nodes are equipped with 2 AMD EPYC 7713 128C 2.0 GHz and 4 AMD MI250 128 GB GPUs each. As the MI250 is a multi-chip module with two GPU dies with access to 64 GB of the overall 128 GB HBM, we regard each MI250 as *two* GPUs and run one MPI process per GPU, i.e., up to 8 MPI processes per node.
- **"Nvidia Grace CPU":** Experiments on the Grace-Grace nodes of the NHR@KIT future technologies (FTP) partition: each node is equipped with 2 Nvidia Grace CPU-Superchips 72C 3.1 GHz, an ARMv9 + SVE2 architecture. We run 8 MPI processes per node with 18 threads per process.
- **"Nvidia GH200":** Experiments on the two Grace-Hopper test nodes of DLR's cluster CARO, equipped with a single Nvidia Grace 72C 3.4 GHz CPU and a single Nvidia Hopper GH200 96 GB GPU each[12]. We run 1 MPI process per GPU, i.e., 1 MPI process per node.

As expected from our weak scaling experiments with real-world data, the number of MPI processes — 1, 2, 4, 8, and 16, except for the GH200 experiments with 1 and 2 only — had less influence on the performance metrics than the chunk size and the hardware; see Figure 11. On every single hardware, runtime and memory behave at least roughly as expected: using the previously defined terminology regarding runtime, if $\mathbb{T}_{\mathrm{LOF}}(s, T, p^2)$ does not decrease further for a smaller chunk size s (e.g., because the problem is then too small to benefit from the available computational resources), then the overall runtime $\sim s^{-1} r^{-1} N \cdot \mathbb{T}_{\mathrm{LOF}}(s, T, p^2)$ is roughly of order s^{-1} as can be seen in Figure 11 for small chunk sizes. For

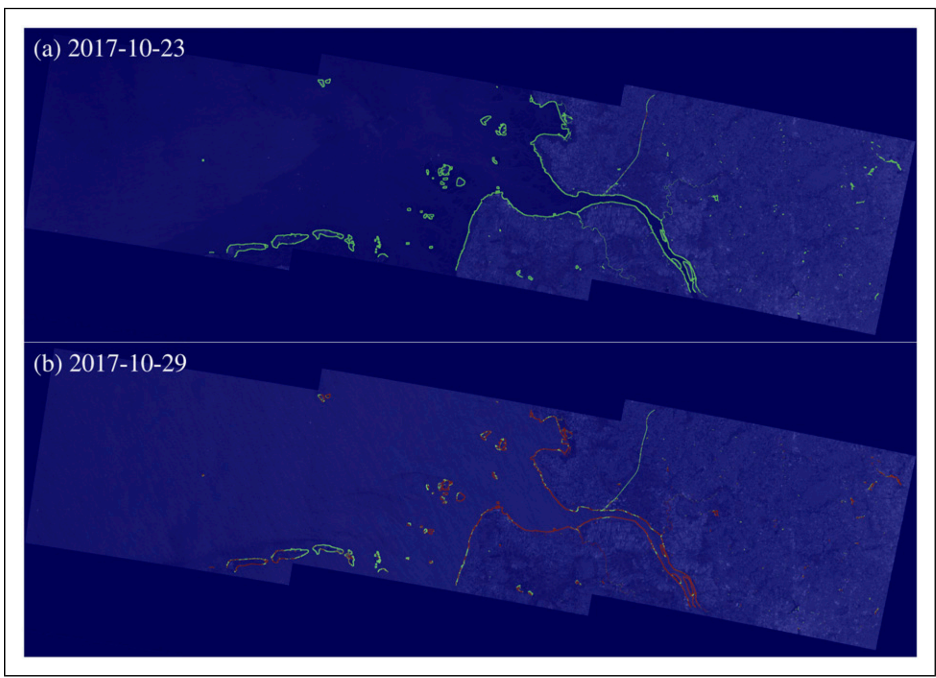

Figure 7. Anomaly map generated with LoOP and a coastline width of 13 px with underlying SAR image for (a) 2017-10-23 and (b) 2017-10-29. Figure (b) shows numerous anomalies (colored in red). Storm Herwart was active on this day.

large s, the memory consumption grows proportionally to s, as indicated by the dotted line representing $\sim Np^2 + sT^2$, as expected.

For the further interpretation of the presented results, it is crucial to point out the two-level structure of Heat's `vmap`-functionality: the upper level is (trivially) MPI-parallel and implemented in Heat, whereas the lower level consists of using PyTorch's own `vmap`[13] on each MPI-process. Hereby, the chunk size argument in Heat's `vmap` is directly passed to PyTorch's `vmap` and thus determines the amount of data processed at once on each process. More precisely, PyTorch's `vmap` is a function transform that batches inputs into internal batched tensors and applies per-operator batching rules. Its performance is determined by the efficiency of the resulting batched operator kernels–of every operation appearing in the function to be vmapped–on the target backend (CUDA, OpenMP etc.) and the respective batched tensor shapes (the first one being determined by the chunk size). Consequently, we may conjecture that the rather strong variation in runtime even among the same hardware type is likely due to different implementation of these low-level kernels for different backends together with the different compute-versus memory-bound-profiles of the respective hardware. In particular, while the calculation of the distance matrix (`cdist`) maps well to dense linear algebra and is well-suited for GPUs, a substantial fraction of the runtime might be driven by data-dependent selection (row-wise `topk`) and irregular memory access (gather-heavy indexing/reductions), which are known to be less suited for GPUs and whose performance can be highly shape- and backend-dependent. One observation supporting this explanatory approach is that LoOP constantly seems to be faster than LOF in our experiments. This is consistent with the algorithmic structure, cf. The Methods section: compared to LOF, LoOP avoids the reachability-distance construction $\max\{k\text{-distance}(y), d(x, y)\}$ and the subsequent neighbor-density look-ups, and instead relies more on reductions (means of squared distances) and element-wise transforms (e.g., $\sqrt{\cdot}$, erf). Consequently, LoOP induces fewer gather-dominated intermediates and probably tends to benefit more from batched execution, which could serve as a plausible, at

Figure 8. Number of anomalies found by each algorithm along the entire German North Sea coast is shown by the blue bars, while the orange dots show the average wind speed. Wind speed values are only shown if they are greater than 30 km/h. The three dates with severe weather events are marked in red.

least partial, explanation of its constantly lower run times.

Summing this up, achieving optimal performance would require extensive experimental tuning of parameters and configurations for every concrete hardware and problem size, or even switching to a customized, problem- and hardware-specific implementation. Nevertheless, the experiments

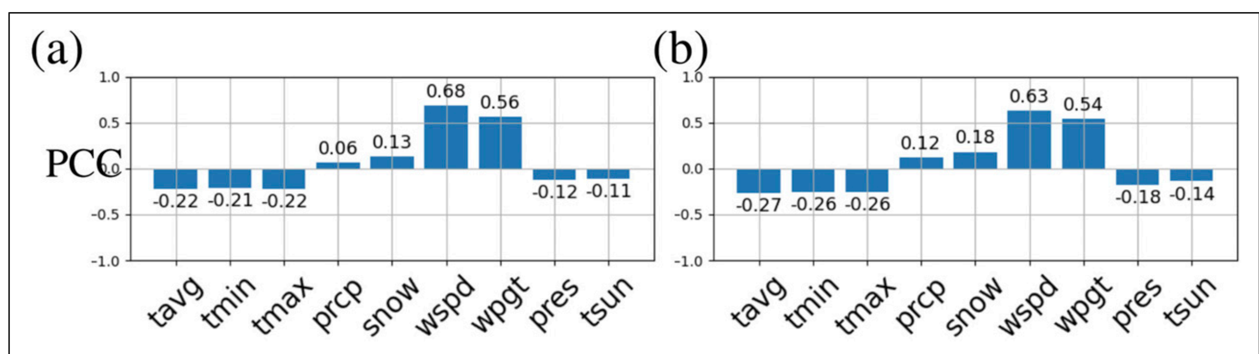

Figure 9. Pearson correlation coefficient (PCC) calculated for the daily number of anomalies found by (a) LOF and (b) LoOP and different weather properties at the German North Sea coast. For the average wind speed (wspd) and maximum wind speed (wpgt) we considered only values greater than 30 km/h or 50 km/h, respectively.

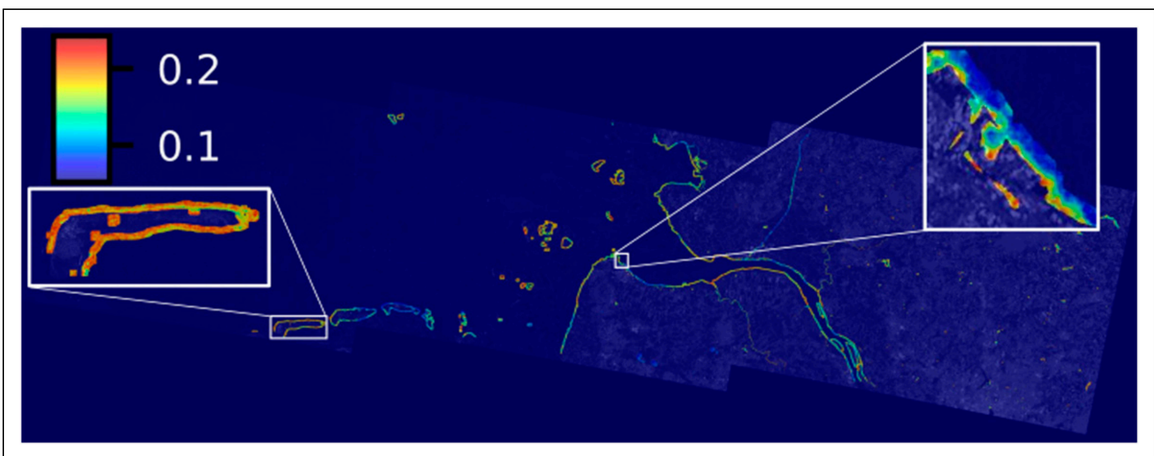

Figure 10. The outlier factor hotspot map generated with LoOP and a coastline width of 8 px shows long-term activity in the German North Sea coast from 2016 to 2023, marking areas of potential erosion.

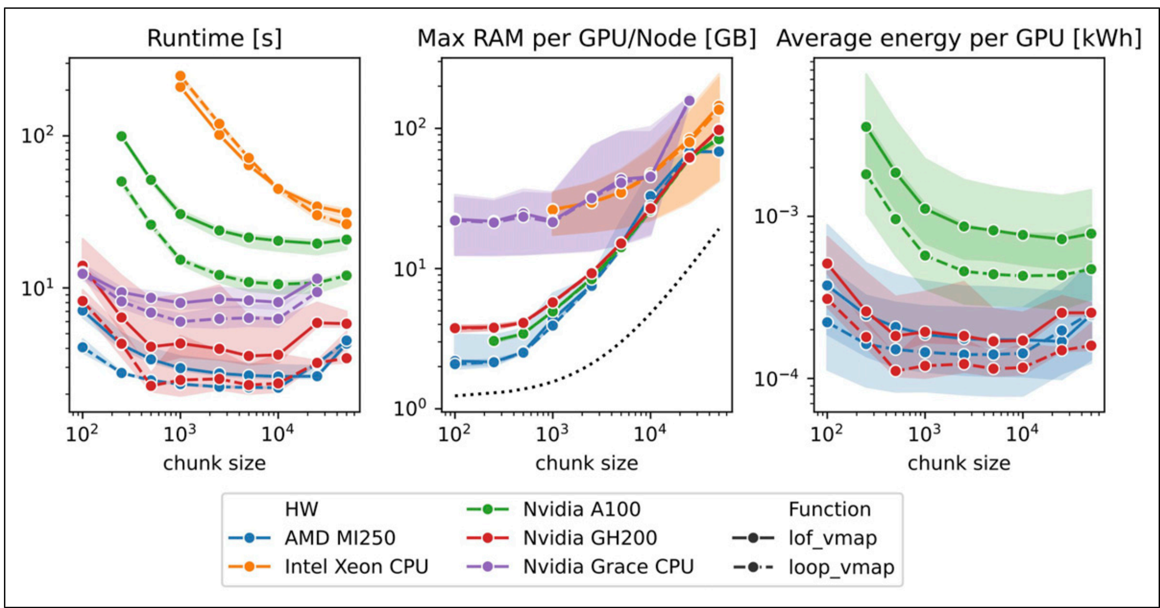

Figure 11. Runtime (lhs), memory consumption (middle), and energy usage (rhs) of vmapped LOF and LoOP on different hardware and with different chunk sizes. Shading indicates the ranges observed for 1 and 2 (Nvidia GH200) or 1, 2, 4, 8, 16 (all other HW) MPI processes. Tracking of CPU energy consumption was not supported by the monitoring methods used in this study. Therefore, these results are only provided for the GPU.

demonstrate that the chosen off-the-shelf approach for up-scaling is–in principle–portable across different vendors (Nvidia and AMD), hardware-types (CPU and GPU), as well as architectures (x86 and ARM).

6. Conclusion and outlook

We demonstrated the feasibility of nationwide, pixel-resolved coastal change detection using multi-year Sentinel-1 SAR data by adapting a patch-wise LOF/LoOP pipeline to Heat's distributed arrays and vmap implementation. Our approach is easy to use and portable across architectures, and additionally achieves near-ideal weak scaling on CPUs/GPUs in our experiments. The method produces date-wise anomaly maps, recognizing severe weather events and hotspot maps identifying regions of recurring aberrations. Reproducible preprocessing and straightforward thresholds make the workflow practical.

Finally, we see the greatest potential for future expansion in four areas. First, improving sensitivity to small-scale

anomalies by moving to higher resolution and richer features is the most direct approach. For example, incorporating 3–10 m SAR/optical data and adding interferometric coherence would be a significant improvement. Second, rigorous validation and benchmarking, such as comparing with erosion measurements directly on the ground. Third, enable an operational input generation for affected communities and first responders. This includes transitioning to near real-time analysis, event-triggered runs, and publishing anomaly and hotspot maps to web services, such as the EO geoservice from DLR (see *Introduction* section). Finally, a future release of Heat is planned that will provide an MPI-parallel implementation of the vanilla LOF to enable usage in situations without localization and longer time series.

Acknowledgements

The SAR images and the water mask have been kindly provided by Paola Rizzoli and Luca Dell'Amore from the Microwaves and Radar Institute, Satellite-SAR-Systems department of the German Aerospace Center (DLR). The authors thank the reviewers for their comments and suggestions that helped to improve the paper.

ORCID iDs

Wadim Koslow https://orcid.org/0000-0003-3912-6615
Fabian Hoppe https://orcid.org/0000-0002-4501-6829
Alexander Rüttgers https://orcid.org/0000-0001-6347-9272

Funding

The authors disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was carried out under the project *Resiliente Versorgungsinfrastruktur und Warenströme im Kontext küstennaher Extremwetterereignisse* (RESIKOAST) by the German Aerospace Center (DLR). The authors gratefully acknowledge the computational and data 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 HPC system CARO is partially funded by "Ministry of Science and Culture of Lower Saxony" and "Federal Ministry for Economic Affairs and Climate Action". Parts of this work were performed on the NHR@KIT Future Technologies Partition testbed funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The contribution of HA has been developed in the project PLan_CV. Within the funding Programme FH-Personal, the project PLan_CV (reference number 03FHP109) is funded by the German Federal Ministry of Research, Technology, and Space (BMFTR) and the Joint Science Conference (GWK).

Declaration of conflicting interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Notes

1. https://geoservice.dlr.de/.
2. https://numpy.org/, https://scipy.org/, https://scikit-learn.org.
3. https://www.dwd.de/.
4. https://meteostat.net/.
5. https://github.com/helmholtz-analytics/heat.
6. Heat's resplit_()-method utilizes MPI Alltoallw and follows the description in Dalcin et al. (2018).
7. https://docs.terrabyte.lrz.de/.
8. This ad hoc choice is based on the authors' experience and was already used in Hoppe et al. (2025).
9. With regard to the difference between allocated and actually used CUDA memory in PyTorch, we call torch.cuda.empty_cache() between each individually considered function to make the maximum memory usage measured by Nvidia SMI during the respective functions consistent with the actual maximum memory usage. Remaining effects due to fragmentation etc. have been minor only in our experience.
10. https://www.dwd.de/DE/klimaumwelt/klimawandel/_functions/aktuellemeldungen/171030_dwd_analyse_zu_sturmtiefs_xavier_herwart.html.
11. https://www.nlwkn.niedersachsen.de/startseite/aktuelles/presse_und_offentlichkeitsarbeit/pressemitteilungen/strandaufspulung-auf-langeoog-beginnt-nach-pfingsten-211986.html.
12. As Heat's DNDarrays currently do not allow to exploit the unified memory of Nvidia's Hopper architecture, we are restricted to the GPU's HBM here.
13. https://docs.pytorch.org/docs/stable/generated/torch.vmap.html [Accessed 02/17/2026].

References

Adesh A, Shobha G, Shetty J, et al. (2024) Local outlier factor for anomaly detection in HPCC systems. *Journal of Parallel and Distributed Computing* 192: 104923. https://doi.org/10.1016/j.jpdc.2024.104923

Almansoori MKM and Telek M (2025) HdLOF: fast, scalable local outlier factor for large-scale, high-dimensional anomaly detection *SoftCOM 2025*, pp. 1–7.

Alshawabkeh M, Jang B and Kaeli D (2010) Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, pp. 104–110. https://doi.org/10.1145/1735688.1735707

Ben-Nun T and Hoefler T (2019) Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. *ACM Computing Surveys* 52(4): 65. https://doi.org/10.1145/3320060

Bradbury J, Frostig R, Hawkins P, et al. (2018) JAX: composable transformations of python+NumPy programs. Available at. https://github.com/jax-ml/jax

Breunig M, Kriegel HP, Ng R, et al. (2000) LOF: identifying density-based local outliers *Proc 2000 ACM SIGMOD Intl*

Conf Manag Data, pp. 93–104. https://doi.org/10.1145/335191.335388

Castro O, Bruneau P, Sottet JS, et al. (2023) Landscape of high-performance python to develop data science and machine learning applications. *ACM Computing Surveys* 56(3): 65. https://doi.org/10.1145/3617588

Corain M, Garza P and Asudeh A (2021) DBSCOUT: a density-based method for scalable outlier detection in very large datasets. *ICDE*: 37–48. https://doi.org/10.1109/ICDE51399.2021.00011

Dalcín L, Paz R, Storti M, et al. (2008) MPI for python: performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing* 68(5): 655–662. https://doi.org/10.1016/j.jpdc.2007.09.005

Dalcin L, Mortensen M and Keyes DE (2018) *Fast Parallel Multidimensional FFT Using Advanced MPI Journal of Parallel and Distributed Computing 128*. DOI: 10.1016/j.jpdc.2019.02.006.

Darema F (2001) The SPMD model: past, present, and future *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, pp. 1–8. https://doi.org/10.1007/3-540-45417-9_1

Götz M, Bodenstein C and Riedel M (2015) HPDBSCAN: highly parallel DBSCAN *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, p. 2. https://doi.org/10.1145/2834892.2834894

Götz M, Debus C, Coquelin D, et al. (2020) HeAT – a distributed and GPU-Accelerated tensor framework for data analytics *2020 IEEE International Conference on Big Data (Big Data)*. London: Nature Publishing Group UK, pp. 276–287. https://doi.org/10.1109/BigData50022.2020.9378050

Gutiérrez Hermosillo Muriedas JP, Flügel K, Debus C, et al. (2023) Perun: benchmarking energy consumption of high-performance computing applications *Euro-Par 2023: Parallel Processing*. Publisher: Springer Nature Switzerland, pp. 17–31. https://doi.org/10.1007/978-3-031-39698-4_2

Hoppe F, Gutiérrez Hermosillo Muriedas JP, Tarnawa M, et al. (2025) Engineering a large-scale data analytics and array computing library for research: Heat. *Electronic Communications of the EASST* 83: 1–26. Available at: https://doi.org/10.14279/eceasst.v83.2626

Koslow W, Rack K, Grabosch TD, et al. (2026) Patch-based anomaly detection on SAR images to localize hotspots on the north and Baltic Sea Coasts. *Remote Sensing Applications: Society and Environment*. Available at: https://doi.org/10.1016/j.rsase.2026.101958

Kriegel HP, Kröger P, Schubert E, et al. (2009) LoOP: local outlier probabilities. In: *Proc 18th ACM Conf Inf Knowl Manag*, 1649–1652. https://doi.org/10.1145/1645953.1646195

Ma H, Hu Y and Shi H (2013) Fault detection and identification based on the neighborhood standardized local outlier factor method. *Ind & Eng Chem Res* 52(6): 2389–2402. https://doi.org/10.1021/ie302042c

Message Passing Interface Forum (2023) MPI: a message-passing interface. Standard Version 4.1. Available at. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

Okkels CB, Aumüller M and Zimek A (2025) On the design of scalable outlier detection methods using approximate nearest neighbor graphs *Similarity Search and Applications*. Springer Nature Switzerland, pp. 170–184.

Paszke A, Gross S, Massa F, et al. (2019) PyTorch: an imperative style, high-performance deep learning library *Advances in Neural Information Processing Systems*, Vol. 32. *NeurIPS 2019*.

Rocklin M (2015) Dask: parallel computation with blocked algorithms and task scheduling *Proceedings of the 14th Python in Science Conference*, pp. 126–132. https://doi.org/10.25080/Majora-7b98e3ed-013

Rüttgers A and Petrarolo A (2021) Local anomaly detection in hybrid rocket combustion tests. *Experiments in Fluids* 62(7): 136. https://doi.org/10.1007/s00348-021-03236-1

Thudumu S, Branch P, Jin J, et al. (2020) A comprehensive survey of anomaly detection techniques for high dimensional big data. *Journal of Big Data* 7(42): 1–30. https://doi.org/10.1186/s40537-020-00320-x

Yan Y, Cao L, Kulhman C, et al. (2017) Distributed local outlier detection in big data *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, pp. 1225–1234. https://doi.org/10.1145/3097983.3098179

Author biographies

Wadim Koslow is a mathematician by training who earned a Master's degree in Mathematics from the University of Cologne. He works in the High-Performance Computing department at the Institute of Software Technology of the German Aerospace Center (DLR). His work focuses on machine learning, particularly anomaly detection.

Fabian Hoppe obtained his Ph.D. in mathematics from the University of Bonn and works as researcher in the High-Performance Computing department at the Institute of Software Technology of the German Aerospace Center (DLR). His research interests include scalable data analytics and ML for science and engineering applications as well as physics-informed AI, uncertainty quantification, combination of traditional numerics and AI/ML, and related software.

Kathrin Rack studied physics at the University of Düsseldorf and holds a Ph.D. in theoretical physics from the University of Cologne for her work in computational biophysics at Forschungszentrum Jülich. Since 2015, she has been supporting the Institute for Software Technology of the German Aerospace Center (DLR), where she was involved in the RESIKOAST research project which focuses on large-scale anomaly detection methods for Earth observation data. Furthermore, she is specialised in high-performance data analytics, machine learning, and software engineering.

Hakan Akdag earned his doctoral degree in theoretical physics from the University of Bonn and works as a software scientist in the High-Performance Computing department at the Institute of Software Technology of the German Aerospace Center (DLR) and as a lecturer at the Cologne University of Applied Sciences. His research interests span, amongst others, scalable machine learning, high-performance data analytics, computer vision, geometric deep learning, and anomaly detection – with a focus on transferring these methods into industrial applications.

Alexander Rüttgers is the team lead of the Scalable Machine Learning research group at the Institute of Software Technology of the German Aerospace Center (DLR). He received his Ph.D. in mathematics from the University of Bonn. His research interests include large-scale machine learning, anomaly detection, and geometric deep learning.

Achim Basermann earned his Ph.D. in Electrical Engineering from RWTH Aachen University. He currently leads the High-Performance Computing department at the Institute of Software Technology at the German Aerospace Center (DLR). His expertise is in performance engineering, high-performance data analytics and compression, and quantum computing.