Supporting Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML Framework

Conference Paper · November 2017

DOI: 10.1145/3144763.3144765

CITATIONS
0

READS
156

4 authors:

Markus Götz
Karlsruhe Institute of Technology
20 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Matthias Book
University of Iceland
136 PUBLICATIONS **559** CITATIONS
SEE PROFILE

Christian Bodenstein
Forschungszentrum Jülich
12 PUBLICATIONS **61** CITATIONS
SEE PROFILE

Morris Riedel
Forschungszentrum Jülich
112 PUBLICATIONS **798** CITATIONS
SEE PROFILE

Supporting Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML framework

Markus Götz
Research Center Juelich
Juelich Supercomputing Center
Jülich, Germany
m.goetz@fz-juelich.de

Matthias Book
University of Iceland
School of Engineering and Natural Sciences
Reykjavík, Iceland
book@hi.is

Christian Bodenstein
Research Center Juelich
Juelich Supercomputing Center
Jülich, Germany
c.bodenstein@fz-juelich.de

Morris Riedel
Research Center Juelich
Juelich Supercomputing Center
Jülich, Germany
m.riedel@fz-juelich.de

ABSTRACT

The development of high performance computing applications is considerably different from traditional software development. This distinction is due to the complex hardware systems, inherent parallelism, different software lifecycle and workflow, as well as (especially for scientific computing applications) partially unknown requirements at design time. This makes the use of software engineering practices challenging, so only a small subset of them are actually applied. In this paper, we discuss the potential for applying software engineering techniques to an emerging field in high performance computing, namely large-scale data analysis and machine learning. We argue for the employment of software engineering techniques in the development of such applications from the start, and the design of generic, reusable components. Using the example of the Juelich Machine Learning Library (JuML), we demonstrate how such a framework can not only simplify the design of new parallel algorithms, but also increase the productivity of the actual data analysis workflow. We place particular focus on the abstraction from heterogeneous hardware, the architectural design as well as aspects of parallel and distributed unit testing.

CCS CONCEPTS

• **Theory of Computation** → Distributed algorithms; • **Information Systems** → Data analytics; • **Computer Systems Organization** → Heterogenous (hybrid) systems; • **Hardware** → Testing with distributed and parallel systems;

KEYWORDS

High Performance Computing, Data Analysis, Architecture Design, Testing

SUPERCOMPUTING'17, November 2017, Denver, Colorado, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Markus Götz, Matthias Book, Christian Bodenstein, and Morris Riedel. 2018. Supporting Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML framework. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, USA, November 2017 (SUPERCOMPUTING'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High performance computing (HPC) is concerned with the coupling of computational resources to enable the solution of large-scale problems in science and engineering. Specific application fields include e.g. simulating the climate in order to forecast the weather, optimizing the flow dynamics of car chassis, or protein folding. While the user domains heavily vary in their methods, in the end they all require the use of some form of software, often developed by the end-users of the application themselves. The development processes and applied engineering approaches for those systems are very different from traditional commercial software. A number of investigations have been performed regarding the reasons. One of the most accurate summaries is given by Basili et al. [5] that is additionally supported by research of Segal and Morris [38] and Schmidberger and Brügge [36].

Their findings can be condensed as follows. First and foremost, the main users of HPC systems are domain scientists. These are experts in physics, chemistry, biology and so forth. This means they often do not have a background in computer science or software engineering, and therefore lack knowledge about engineering approaches and their usefulness. More importantly, their main objective lies in the domain science and not in engineering code. As a result, technical process optimization and development methods take a back seat compared to the actual scientific question. This also makes requirements analysis challenging, as most of the features are often unknown at design time, and are changing heavily in the process [38]. This has led to an implicit adoption of an agile development process, albeit without following any formal methodology.

Second, the technological challenges in HPC are enormous. The systems themselves are highly parallel, and designing algorithms for them is a time-consuming activity. Most of the legacy code is

therefore written in low-level programming languages like Fortran and C, which continue to be used to this day, as they allow various optimizations and access to accelerator hardware like coprocessors. Changing this infrastructure seems unlikely as it would require rewriting highly complex software with decades worth of fine-tuning. In lieu of that, new technologies can only be slowly adapted and integrated. The latest developments include for example the broader use of C++ and its object-oriented programming model as well as the use of scripting languages (mainly Python) as interface wrappers to simply application coding [36].

Finally, the ranking of engineering goals in HPC differs from the engineering of information systems. Carver et al. [9] point out that first and foremost the correctness of the code matters, followed by performance/scalability, portability and maintainability, in that order. However, the verification and validation of code in HPC is challenging due to the complexity of the application domains and especially with respect to the cost of a potentially validating experiment [5]. Structured software testing has therefore become frequently used in HPC recently, but is still only used to a limited degree. Testing distributed and parallel systems is not well studied, much less supported by tools. Performance gains are usually invested in increasing the resolution and number of parameters of a simulation, rather than shortening the time to obtain a solution. Portability aspects have to be considered due to the quickly changing nature of the execution hardware. The lifetime of an HPC system is often in the three to five year range, while low-level libraries are around for decades. The low importance assigned to maintainability is often reflected in poor code quality, little to no documentation, heavy code duplication and other practices often frowned upon in other application domains. The low maintainability can also be attributed to the workflow of HPC application development. While major base libraries are well maintained, a lot of the application code is developed in a trial-and-error fashion to test models, and is more intended to be a throw-away prototype. Working code, though, is often not refactored or redesigned from scratch, but directly taken as a foundation for further development.

Currently, the HPC community sees the rise of a new sub-field—data analysis and machine learning. While these techniques have always played a role in the experimentation since beginning of HPC, large-scale data intensive experiments, such as the TOAR climate research database [37], the earth science data repository PANGAEA [16] or the planned Square Kilometer Array [15] have recently led to a steep increase in interest and requirements. Typical analysis goals are the unsupervised identification of patterns in data, the classification of observations into groups or the detection of anomalous readings. While these goals do not differ from the small-scale data analysis world, the sheer amount of data and its bandwidth require the use of HPC resources. Current engineering research focuses on the parallelization of algorithms, their scalability with respect to the number of data items, and the precision of predictions. Moreover, there is an increasing desire to couple simulations with in-situ data analytics [43], underlining the need for an integrated framework.

Since the broad development of HPC-driven data analysis and machine learning applications is still a relatively young field with little existing code, there is an opportunity to establish the use of suitable software engineering practices from early on. In this paper, we discuss the typical data analysis workflow on HPC systems with respect to reappearing patterns, reusable components such as standard analysis algorithms, as well as aspects of application development. As a tool to support the adoption of software engineering practices such as modular design and structured testing, we introduce the Juelich Machine Learning Library (JuML), an HPC data analysis framework that implements said techniques. Its five major design requirements are: (1) Provision of scalable machine learning algorithms, (2) transparent execution on different hardware backends, (3) well-documented, tested and intuitive API, (4) support for the scripting workflow of domain scientists and (5) a pluggable architecture design to enable the framework extension. JuML has proven to be successful in a number of use cases, one of the land cover type classification, presented in Section 5.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of modern, heterogeneous HPC systems and technologies. Related work on software engineering efforts in the HPC field as well as distributed, scalable data analysis frameworks is discussed in Sect. 3. In Sect. 4 the typical data analysis workflow on HPC systems is discussed and pertinent design solutions of JuML are presented, including how applications can be tested. After the land cover type detection use case study in Sect. 5, Sect. 6 summarizes our contributions and points out opportunities for future work.

2 BACKGROUND—ANATOMY OF HIGH PERFORMANCE COMPUTING SYSTEMS

HPC systems are very diverse in their hardware components and properties, and each can be considered more or less unique. In addition to that, particular product brands tend to disappear once market adoption is reached, due to vendors introducing new marketable platforms, leading to numerous code changes for developers relying on the a particular system design. Nowadays systems are mostly clusters with basic compute nodes equipped with a multi-core processor and shared main memory. Additionally, larger systems, tend to be designed in a modular or heterogeneous fashion, meaning that the system includes specialized coprocessors, such as for example general-purpose graphics cards (GPGPUs) [31], field-programmable arrays (FPGAs) [33] or Many Integrated Cores (MICs) [22] boards. The range of choices is large and every single candidate requires a separate programming model and often special tailoring of the code to efficiently execute on said hardware. This highly increases development effort and cost, while at the same time reducing portability. There are efforts to abstract from the peculiarities using high-level APIs, e.g OpenCL [42], but they need strong compiler support or still significant code adjustments. High performance programs usually run in a single-program, multiple data items (SPMD) fashion. That means each of the nodes executes the exact same binary and only works on a different part of the simulated space or data partition. The de facto standard inter-process communication framework for this on HPC systems is the Message Passing Interface (MPI) [20], which not only automatizes the distributed and parallel spawning of the processes, but also provides the message exchange primitives. These primitives usually allow to send and receive arrays of single data types synchronously or asynchronously in a point-to-point or collective manner.

3 RELATED WORK

Over the past years, there have been increased efforts to address the low prominence of software engineering techniques in HPC that we summarized in Sect. 1. One of the major projects in this area has been DARPA's HPCS lighthouse effort [25], starting already in 2002, to not only improve the machines' hardware capabilities, but also the software landscape to increase productivity. In its wake, a number of other efforts have been established to foster software engineering in HPC. Among these are works on the usage of software engineering tools and methods in HPC [29], the usage of agile development processes [40] or studies of performance in comparison to maintainability and scalability [34]. Moreover, there are dedicated software engineering teams in simulation projects such as the HPC-SE team at the Barcelona Supercomputing Center or the SimLabs in Juelich [3]. The awareness for software engineering methods is slowly arriving in the application domains as well.

In the data analysis area, we can currently observe a potpourri of tools, frameworks and libraries claiming to enable scalable and large-scale experimentation. At the lowest level, there are Theano [8], TensorFlow [1] and Arrayfire [26]. These are efficient matrix and tensor computation libraries, capable of exploiting CUDA and OpenCL capable devices respectively, and widely used in the area of data analysis and machine learning. On top of them, there are higher-level libraries dedicated to neural networks and machine learning. Among them are for example Caffe [23], (py)Torch and MXNet [11] and even support multi GPGPU learning.

However based on their design and the support for traditional unsupervised approaches, MLPack [14] and Intel DAAL [22] are the most similar to JuML. Both of them offer efficient algorithm implementations and the later also a computation distribution strategy. For an application developer, the object-oriented design of both libraries assists the quick development of analysis tools that exploit parallelism in C++. Despite their specific strong suits, both are subsets compared to what JuML is aiming to achieve.

MLPack puts a strong emphasis on code correctness, and therefore unit testing, bug tracking and performance analysis. Its unit test suite is comprehensive and covers most of the code, but is only executed single-threaded. The performance tests measure algorithm analysis qualities, such as prediction accuracy, as well as implementation performance like execution time and memory consumption as part of the build process. Comparability of these measurements is not given as they measure raw performance instead of relative performance increases.

DAAL, instead, also allows the development of data analysis applications in Java and Python, using the included bindings. It has a built-in notion of algorithm parallelization across multiple distributed nodes and supports the use of Intel's MIC Xeon Phi. However, the actual data distribution implementation included in DAAL works only in conjunction with Apache Spark or Hadoop as the parallel processing platform. For technical reasons, such as scheduler and file system incompatibility, MPI and Spark/Hadoop will often not be used on the same cluster systems due to difficult integration [18]. The data distribution code with MPI as communication framework needs to be provided by the application user [2], which is often the most error-prone and time-consuming part of HPC application development.

4 THE JUELICH MACHINE LEARNING LIBRARY FOR DATA ANALYSIS IN HIGH PERFORMANCE COMPUTING

The general data analysis workflow on HPC systems does not differ in its essentials from small-scale analysis. There are a number of standard processes explained in the literature and established in the industry, such as KDD or CRISP-DM [4]. Despite minor differences in these, the main steps of the analysis process always are: 1. data selection, 2. characteristics exploration and identification, 3. preprocessing, 4. model construction according to the analysis objective, 5. evaluation, 6. postprocessing, and 7. deployment and preservation of analysis results. Even though described in a very linear fashion, the actual process is not rigid, but rather iterative in nature, with the most cycles between step 3 and 6.

Common analysis goals are the classification of data items into categories, detection of recurring patterns, prediction of future values or filtering of outliers. This analytical framework is well understood from small-scale data analysis and can simply be used as a set of standard algorithms in the HPC community. What differs is the data volume and number of observations to be analyzed. These can easily exceed a number of terabytes up to some petabytes for a single problem. The framework should take care of the data distribution as well as algorithm parallelization while supporting each of the data analysis workflow steps described above.

The open-source Juelich Machine Learning Library (JuML) [21] strives to implement such a framework. It is written in C++ and uses ArrayFire as its computation engine to support OpenCL-capable coprocessors as well as CUDA-capable GPGPUs. JuML aims at supporting data analysis application developers and parallel algorithm designers with each of the above steps of the workflow. For this, JuML is designed in a modular fashion with generic, reusable components designed for application and framework developers. This reduces code duplication and introduces single entities for optimization. Moreover, it features a high-level API that allows the transparent definition and assignment of parallel processing resources for individual computation steps. Each of the components is designed with the goal of speeding up data analysis in a distributed HPC system with MPI as the distributed, parallel processing and message exchange platform. Figure 1 depicts an overview of JuML's internal architecture. Generally, it is divided into two virtual parts. On the one hand, there are the classes and APIs meant to be used by the application developers, which are situated at the top. These are kept abstract and high-level in order to hide parallelization details from the analysts, while on the other hand, the low-level routines are aimed at simplifying distributed and parallel analysis algorithm development. In the following subsections, we will highlight some of the most important engineering issues in building HPC data analysis applications, and corresponding solution strategies implemented or supported in JuML.

4.1 Data Access and Distribution

The central unit of each analysis step is the dataset that is being investigated. For many use cases, the amount of data is so large that it needs to be split up and distributed across a number of independent nodes. While the access pattern is often arbitrary for simulations, e.g., particles in a certain spatial cell, it is regular for data analysis

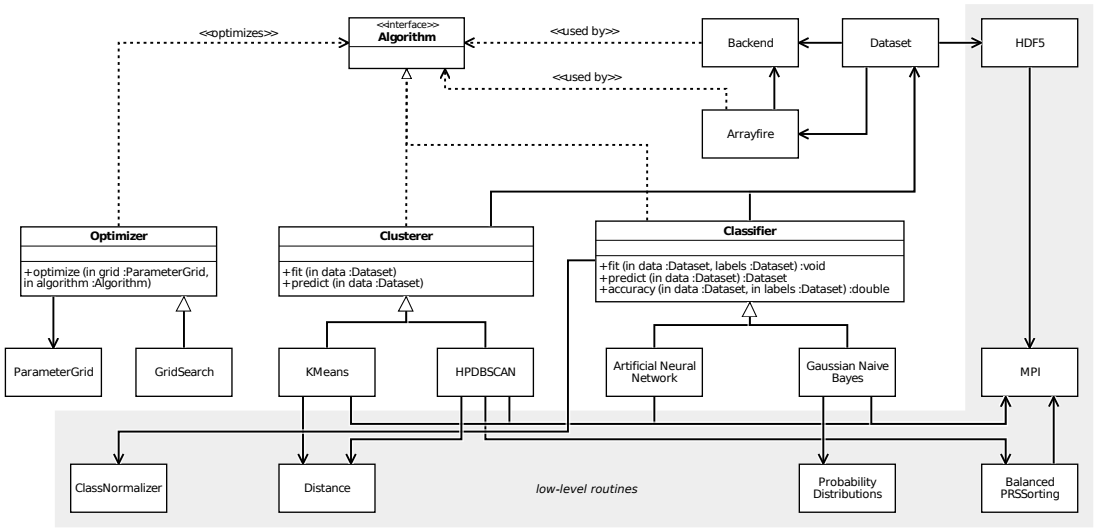


Figure 1: UML diagram of JuML's architecture.

uses cases. Each of the nodes receives an equally sized chunk of the data in order to provide the most optimal load balancing and thus peak parallel performance. Sometimes a halo is required, i.e. an overlap in the data chunks, that allows the merging of partial results of parallel computations. Essentially, this means there are only two major data access strategies, or four, if one includes additional weights, that account for performance differences of the allocated processors and coprocessors.

```
1  juml::Dataset data("/home/analysis_data.h5", "samples");
```

Listing 1: Distributed dataset access example.

Based on this, the distribution strategies can be encapsulated into reusable entities. In JuML this is realized in the `Dataset` class, which abstracts the highly complex parallel I/O implementation details from the user. Instead, the user only needs to know the path to the desired data file and the name of the request dataset in the file, and pass both to a `Dataset` constructor. Listing 1 shows an example. JuML's `Dataset` object is implemented in a lazy loading fashion, which means data is only loaded if really required. This has two advantages: on the one hand, it increases parallel I/O performance by not loading superfluous data, and on the other hand, it hides the actual chosen data distribution strategy. A concrete data analysis implementation analyzing a dataset knows best which distribution strategy it requires. Therefore, it will chose at analysis time one of the four distribution strategies explained above and actually request the `Dataset` object to fetch data from the storage system.

This approach and the interface design is similar to Resilient Distributed Datasets (RDDs) in Apache Spark [28] and highly simplifies development efforts for application developers. For JuML framework developers, it centralizes I/O code and thereby enables focused performance tuning, having to enhance only one entity, and the easy extension of other distribution strategies, if required. JuML currently allows users to load and store data in the parallel data format HDF5 [17] and the equi-chunking strategy. Currently, netCDF support, another parallel data format, and the halo-chunking strategy are in development. Redistribution of the data stored in a `Dataset` is not supported by design as it introduces a significant performance bottleneck. In summary, JuML's `Dataset` implementation specifically supports HPC data analysis application developers in steps 1 and 7 of the data analysis process cycle.

4.2 API Design

JuML's API is generally designed in a way to resemble the APIs of well-known single-threaded, single-node data analysis libraries, e.g., scikit-learn [32] or SHOGUN [41]. This should simplify porting small-scale data analysis code to HPC systems, if required for the application use cases and makes it easier for new users feel to familiar with the JuML framework. The individual components are modularized into individual entities that each model one particular data analysis method or algorithm. Each of these entities accepts `Dataset` objects as input and equally generates a `Dataset` as output. At the same time, all of the analysis algorithms implement a common interface. Ultimately, this allows data analysis application developers the easy and transparent exchange of the chosen analysis algorithm and therefore minimizes required manual code changes.

In contrast to the APIs of scikit-learn or SHOGUN, though, JuML always requires two additional experimentation parameters beside the actual analysis parameters. These are a handle for the local computation backend, e.g., CPU, GPU etc., and a handle for the cluster nodes on which to run the data analysis algorithm. The local computation backend choice is simply forwarded to ArrayFire, which in turn deploys the computation kernels correctly. For the global parallelization strategy, that is the selection of nodes, JuML accepts a MPI communicator, a data structure of the MPI framework that encapsulates a set of computation nodes. These two handles are sufficient for any JuML data analysis algorithm to completely parallelize the data analysis. For an application developer, this drastically reduces the amount of code that needs to be written and the parallelization knowledge required. Where before, he would have

to implement the communication code manually (which accounts for a major share of lines of code in HPC applications), the same is now expressed by two singular values. Listing 2 shows an API usage example of an arbitrarily selected data analysis algorithm that computes locally on the GPU and uses the entire available node allocation of the cluster system.

```
#include <juml.h>
#include <mpi.h>

int main(void) {
    juml::GaussianNaiveBayes gnb(
        juml::Backend::GPU, // local gpu backend
        MPI_COMM_WORLD // select global node allocation
    );
    return 0;
}
```

Listing 2: C++ API usage example—creation of a GNB classifier computing on GPUs and all available nodes.

In addition to that, JuML also supports the usage of Python as a scripting language. This shall ease the transition of data analysts coming from the small-scale data analysis world, where the scripting language is broadly used, to the large-scale data analysis world. For this, JuML employs the technique of automatic code generation. Specifically, the SWIG [6] interface generator is utilized to accomplish this task. It automatically searches JuML's C++ sources, identifies classes and generates matching Python classes.

In principle, it is also possible to generate bindings for other languages with SWIG, say R or Julia, but this should be treated with care. Each of these languages have their own approach to working in data analysis and are strongly focused on particular data containers. A wrapper needs to carefully support these differences in the approaches to maintain the possibility of a smooth adaption of JuML. As of the time of this writing, Python is the most widespread data analysis scripting language in the HPC environment and is therefore so far the only supported other programming language despite C/C++. In summary, JuML's API design shall mainly assist HPC data analysis application developers in steps 4 and 5 of the data analysis process. Listing 3 shows an example on how to use the generated Python bindings using the example of a Gaussian Naive Bayes classifier introduced above.

```
import juml
from mpi4py import MPI

gnb = juml.GaussianNaiveBayes(
    juml.Backend.CPU, # local cpu backend
    MPI.COMM_SELF # global node allocation
)
```

Listing 3: Python API usage example—instantiation of a GNB classifier computing on CPUs and a single node only.

4.3 Reusable Components

JuML also offers a number of reusable components that are not meant for application developers but rather algorithm developers. Among these are for example class label normalizers, distance functions, probability density accumulators and more. As an example, we will discuss the distributed parallel sorting algorithm that is one of the major components required for parallelizing a number of data analysis algorithms.

Distributed parallel sorting is a key to domain decomposition in data analysis algorithm implementations. In simulation codes one can often find a natural object or systems that allows to split up the domain into independent sub-problems. These in turn can then be assigned to individual processes and computed in parallel. This heavily reduces the amount of communication and limits synchronization to the sub-problem boundaries. The major benefit is a highly scalable and more optimal parallel computation. A typical example is the subdivision of a simulated fluid dynamics space into sub-volumes. For data analysis problems, however, this can not be as easily done, since there is no inherent divisible system in the domain except the spanning boundary of each of the analysis features and their minimum and maximum. If this space is subject to some ordering, for instance a partial one, it is possible to perform a data-driven domain decomposition. In order to impose such an ordering, one needs to sort the data. Afterwards, the resulting independent data chunks can be assigned to processors as in the simulation problems.

Among the parallel data analysis algorithms that use this strategy are for example distributed decision trees [7] or the parallel HPDBSCAN [19] algorithm. JuML provides an optimized version of such a global sorting algorithm—that is, the processor with the lowest rank has the smallest element and the one with the highest rank has the largest element, and every element in-between is partially ordered—in the form of the *balanced parallel sorting by regular sampling* [39]. This algorithm is highly scalable to large amounts of data and auto-balances the number of data items each processor receives.

Using this reusable framework, the amount of code and development time required to implement new data analysis algorithms in the JuML framework is greatly reduced and the probability of implementing a highly scalable solution increased. Since application use case developers also often become framework developers on HPC systems, as explained in Sect. 1, this is indirectly also a benefit for the application development use case, where JuML does not yet provide an implementation of the required data analysis method. Depending on the application development stage, JuML's reusable components support the development of HPC data analysis applications in steps 2 to 6 of the data analysis process.

4.4 Testing

Testing, specifically unit and system testing, is a difficult topic in the high performance community. There are a number of testing goals beside simple correctness, which are not widely considered in other software development areas, such as numerical stability of the computations, a multitude of different hardware environment configurations—CPU, GPGPUs, FPGAs, etc.—as well as a high degree of parallelization and concurrency. There are works by various authors on how to effectively tackle these generally [36] and in details through the application of effective testing methods [29], mocking approaches [12] or concrete case studies [30]. The main issue is the diffusion of the findings into the practical application within the HPC projects, where often one can find little to no use at all.

Most HPC projects that actually do test their code make use of standard unit testing frameworks such as GTest [44] or Boost Test. However, the way that these tests are usually designed and

executed is flawed. First, the tests are either not provided for the parallel and distributed sections of the code. Second, these either test only low-level functionality, say a singular computational kernel, or they are not executed in parallel. This means that tests actually often do not cover the most complex and critical aspects of the code. The two main reasons for this are the lack of parallel and distributed testing frameworks and the unwillingness of application developers to commit expensive and limited compute resources for "use case irrelevant" computation. Third, if tests are provided, they are usually replicated and slightly adjusted for each of the computation backends, e.g., CPU, GPU, etc. Considering good software engineering practices, this violates the don't-repeat-yourself (DRY) principle [45].

JuML tries to overcome these problems by providing the possibility of performing parallel and distributed unit tests executed on each computational backend. These are not only intended for the JuML framework developers, but are also accessible for application developers. The test execution on the different computation backends is realized by extending the test case definition macros of JuML's baseline testing framework GTest. Instead of defining test cases using the `TEST` macro, a JuML developer should use `TEST_ALL`, which generates an individual test case for each of the automatically detected and set up computation backends. This way, tests need to be written only once.

```
// original fixture class of Google Test
class FIXTURE_TEST {
FIXTURE_TEST() {
// setup code here
}
}

// forward definition inheriting from the fixture
#define INTECEPTOR_FORWARD_DEFINITION(FIXTURE) \
class FIXTURE##_Interceptor : public FIXTURE { \
protected: \
void test(); \
};

// per-backend test generator
#define TEST_BACKEND_F(FIXTURE, BACKEND) \
TEST_F(FIXTURE##_Interceptor) { \
// set backend
juml::setBackend(BACKEND); \
// call the test case
this->test(); \
}

// definition of the main macro for all backends
#define TEST_ALL_F(FIXTURE) \
TEST_INTERCEPTOR_FORWARD_DEFINITION(FIXTURE) \
TEST_BACKEND_F(FIXTURE, juml::Backend::CPU) \
#ifdef OTHER_BACKEND \
TEST_BACKEND_F(FIXTUREm OTHER_BACKEND) \
#endif \
void FIXTURE##_Interceptor::test


TEST_ALL_F(FIXTURE_TEST, FEATURE) {
// test code here
}
```

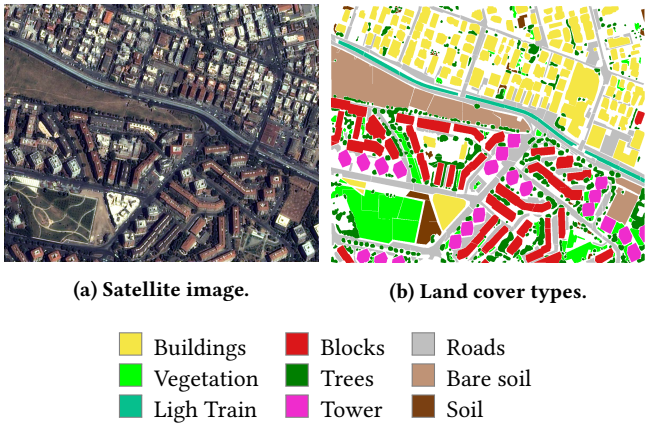
Listing 4: `TEST_ALL_F` macro implementation sketch.

Moreover, there is also a variant that allows the developer to utilize test fixtures. This is particularly interesting for testing data analysis code as it usually requires to load a particular test data set on the which correct analysis is tried. For this purpose, JuML offers an additional `TEST_ALL_F` macro for unit test cases with fixtures. While the implementation of `TEST_ALL` is straightforward, the latter is not. It requires the generation of an additional external fixture class, similar to what Google Test is doing behind the scenes, in order to encapsulate the data generation. Unlike the generated Google Test cases, however, the call needs to be intercepted and the correct computational backend set beforehand. Therefore JuML needs to mimic this behavior and assign it accordingly. Listing 4 shows an algorithmic sketch of how the `TEST_ALL_F` macro has been implemented.

Furthermore, JuML provides an extension to the test runner CTest [27] that is uses to execute the tests distributed on the cluster system. The added `ADD_MPI_TEST` function again registers an individual test for each of the used node counts, which enables better error tracing. In this function, common problematic edge cases are checked, such as a node count that is a prime number or the usage of just a single core. Application developers can override these node allocations and provide their own tested node counts at any time. Generally, JuML's testing tries to keep the number of tested nodes low, in order to preserve computation time of project allocations. In addition to that, there are also a number of standard data sets already included in the JuML bundle, such as Fisher's iris data set that are not only useful for code correction tests, but also to benchmark freshly implemented new data analysis methods.

5 USAGE IN PRACTICE

JuML is already used in a number of scientific research projects. As a first case study demonstrating JuML's benefits, we describe here a remote sensing problem that employs JuML's artificial neural network (ANN) implementation in order to perform land cover type classification [35]. This means that a probabilistic model is constructed that can classify each pixel of a satellite image, as seen in Figure 2a, according to its land cover type, e.g., field, road, building, etc. To achieve this, a neural network learns patterns from annotated ground truth data and should then be able to detect said patterns in new, unseen data. With such a neural network, it is possible to automatically generate parts of street maps or monitor urban planning efforts. Figure 2b depicts an example from the city of Rome analyzed in this way to predict land cover types.



(a) Satellite image. **(b) Land cover types.**

Figure 2: Example of a remote sensing land cover type prediction problem. Aerial image of Rome with a geometrical resolution of 1.3 m and 55 different frequency bands.

From the domain scientists' point of view, i.e., the remote sensing experts, JuML has greatly helped in providing faster experimentation, while keeping the code similar in complexity compared to single-threaded implementations. If the exact same neural network would have been implemented in Python using state-of-the-art deep learning frameworks, such as Keras or Lasagne, the actual analysis code would have been very similar in length (off by less than ten lines of code). However, those frameworks can only facilitate a single GPU on a single node. JuML instead allows the distribution of the computation across multiple nodes simply by passing an additional argument, the MPI communicator, to the algorithm. Using this approach, the experimentation computed much faster: using eight processing nodes yielded a speed-up of five, or in other words, only one fifth of the time was required to obtain the solution. With an overall prediction accuracy of ~91, 1% the land cover types have been predicted mostly accurately achieving comparable results to other recent studies, such as for example Cavallaro et al. [10]. The experimentation was performed on the JURECA supercomputer [24] using multiple GPGPU compute nodes, each having two CUDA-aware [31] NVIDIA K80s. Figure 3 shows the obtained speed-ups during the training phase with a batch size of 100.

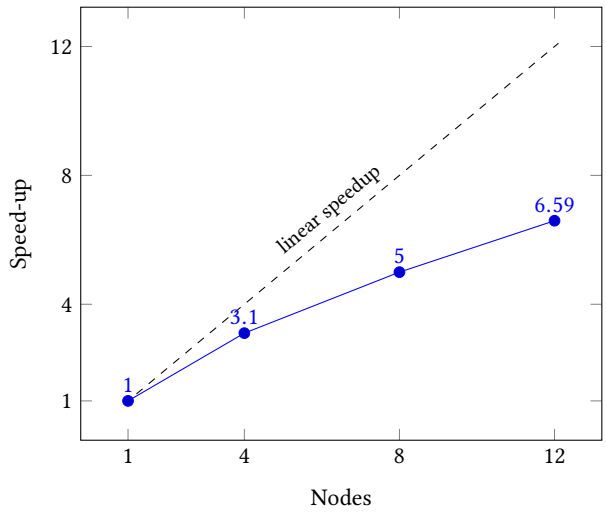
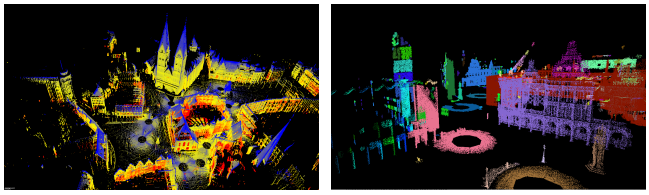


Figure 3: ANN Speedup with 1000 hidden neurons on the Rome land cover type classification problem.

Another example for JuML's application is the benchmarking of the data analysis hardware module of the experimental DEEP-EST supercomputing system [13]. One of the project's use cases will perform object detection and segmentation in multi-dimensional point clouds. These are spatial coordinate meshes of electro-magnetic wave reflections off surfaces, usually recorded by autonomous LiDAR vehicles or drones. Analysis goals include the identification of buildings and structures that can then be used for the generation of maps or city planing change tracking. An example of such a four-dimensional point cloud of the old town of Bremen, including already segmented objects, can be found in Figure 4. JuML supports the implementation of the use case in two major ways. First, it provides all required low-level routines that are needed for the main analysis algorithm, i.e. HPDBSCAN [19], to be ported to the new platform. Second, and even more important, it also significantly assists in scaling the analysis application to larger data scales. The project plan is to not only investigate a single town, but to analyze the point cloud of the entire nation of the Netherlands. The transition in this case is going to be transparent if realized with JuML, as it simply requires exchanging the data path origin. Our library then takes care of the correct data distribution using the aforementioned `Dataset` class. This will drastically reduce the amount of code having to be written, thus reduces sources for errors and speeds up development. Due to the fact that the DEEP-EST project is still underway, we are unfortunately not yet able to show any scalability or speed-up plots at this time.



(a) Raw data. **(b) Segmented objects.**

Figure 4: Example of four-dimensional point cloud data of the old town of Bremen. The four dimensions are the spatial coordinates and the heat radiation off the surface.

6 CONCLUSION AND FUTURE WORK

In this work, we have recapitulated the lack of thorough application of good software engineering practices in building high performance community applications. With the emerging field of large scale data analysis arises the opportunity to employ software engineering approaches right from the start. We have therefore introduced the HPC data analysis library JuML, that enables easy encapsulation of data access and distribution can easily be encapsulated and provides common interfaces for algorithm classes that allow a simple and high-level definition of a scalable parallalelization strategy for application developers. Reusable low-level components like global sorting routines or class-normalizers enable the effective implementation of additional library features, such as new analysis algorithms, by library developers. The incorporation of coprocessors in heterogeneous cluster systems can be achieved via the hardware abstraction technology ArrayFire, serving as JuML's computation engine. This made it a prime for usage in various research projects, such as for example the benchmarking suite for data-analysis module of the experimental DEEP-EST system, as well as the presented land cover classification use case. In the latter, we have achieved a peak speed-up of up 6.59 using 12 graphic cards, while maintaining the same code length compared to non distributed state-of-the-art libraries.

Contrary to other fields, the question of performance testing is of paramount interest for the HPC community. In our future work, we will therefore strive to find the major performance tuning parameters and try to abstract them in some form of hardware abstraction

layer. The important measurement metric here is the parallel efficiency, which measures whether a scalable implementation can maintain its execution time.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Ryo Asai. 2016. Introduction to Intel DAAL, Part 2: Distributed Variance-Covariance Matrix Computation. https://goparallel.sourceforge.net/wp-content/uploads/2016/04/Colfax_Introduction_to_Intel_DAAL_2_of_3.pdf. (2016).

[3] N Attig, R Esser, and P Gibbon. 2008. Simulation Laboratories: An innovative Community-oriented Research and Support Structure. *CGW* 7 (2008), 1–9.

[4] Ana Isabel Rojão Lourenço Azevedo and Manuel Filipe Santos. 2008. KDD, SEMMA and CRISP-DM: a parallel overview. *IADS-DM* (2008).

[5] Victor R Basili, Jeffrey C Carver, Daniela Cruzes, Lorin M Hochstein, Jeffrey K Hollingsworth, Forrest Shull, and Marvin V Zelkowitz. 2008. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. *IEEE software* 25, 4 (2008), 29.

[6] David M Beazley et al. 1996. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++.. In *Tcl/Tk Workshop*.

[7] Yael Ben-Haim and Elad Tom-Tov. 2010. A Streaming Parallel Decision Tree Algorithm. *Journal of Machine Learning Research* 11, Feb (2010), 849–872.

[8] James Bergstra, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, Ian Goodfellow, Arnaud Bergeron, Yoshua Bengio, and Pack Kaelbling. 2011. Theano: Deep learning on gpus with python. (2011).

[9] Jeffrey C Carver, Richard P Kendall, Susan E Squires, and Douglass E Post. 2007. Software development environments for scientific and engineering software: a series of case studies. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. Ieee, 550–559.

[10] Gabriele Cavallaro, Mauro Dalla Mura, Edwin Carlinet, Thierry Géraud, Nicola Falco, and Jón Atli Benediktsson. 2016. Region-based classification of remote sensing images with the morphological tree of shapes. In *Geoscience and Remote Sensing Symposium (IGARSS), 2016 IEEE International*. IEEE, 5087–5090.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[12] Thomas Clune, Hal Finkel, and Michael Rilee. 2015. Testing and debugging exascale applications by mocking MPI. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. ACM, 5–8.

[13] European Commision CORDIS. 2017. DEEP-EST. http://cordis.europa.eu/project/rcn/210094_en.html. (2017).

[14] Ryan R Curtin, James R Cline, Neil P Slagle, William B March, Parikshit Ram, Nishant A Mehta, and Alexander G Gray. 2013. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research* 14 (2013), 801–805.

[15] Peter E Dewdney, Peter J Hall, Richard T Schilizzi, and T Joseph LW Lazio. 2009. The Square Kilometre Array. *Proc. IEEE* 97, 8 (2009), 1482–1496.

[16] Michael Diepenbroek, Hannes Grobe, Manfred Reinke, Uwe Schindler, Reiner Schlitzer, Rainer Sieger, and Gerold Wefer. 2002. PANGAEA—an information system for environmental sciences. *Computers & Geosciences* 28, 10 (2002), 1201–1210.

[17] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A File Format and I/O library for High Performance Computing Applications. In *Proceedings of supercomputing*, Vol. 99. 5–33.

[18] Richard Gerber, William Allcock, Chris Beggio, Stuart Campbell, Andrew Cherry, Shreyas Cholia, Eli Dart, Clay England, Tim Fahey, Fernanda Foertter, et al. 2014. *DOE High Performance Computing Operational Review (HPCOR): Enabling Data-Driven Scientific Discovery at HPC Facilities*. Technical Report. Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US).

[19] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: Highly Parallel DBSCAN. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2.

[20] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Vol. 1. MIT press.

[21] Markus Götz, Christian Bodenstein, Phillip Glock, and Matthias Richerzhagen. 2017. Juelich Machine Learning Library. https://github.com/FZJ-JSC/JuML/. (2017).

[22] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[24] Jülich Supercomputing Centre. 2016. JURECA: General-purpose supercomputer at Jülich Supercomputing Centre. *Journal of large-scale research facilities* 2, A62 (2016). https://doi.org/10.17815/jlsrf-2-121

[25] Jeremy Kepner. 2004. HPC productivity: An overarching view. *The International Journal of High Performance Computing Applications* 18, 4 (2004), 393–397.

[26] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. 2012. ArrayFire: a GPU Acceleration Platform. In *SPIE Defense, Security, and Sensing*.

[27] Ken Martin and Bill Hoffman. 2010. *Mastering CMake*.

[28] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, et al. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[29] Hoda Naguib and Yang Li. 2010. (Position Paper) Applying software engineering methods and tools to CSE research projects. *Procedia Computer Science* 1, 1 (2010), 1505 – 1509. https://doi.org/10.1016/j.procs.2010.04.167 ICCS 2010.

[30] Aziz Nanthaamornphong. 2016. A case study: test-driven development in a microscopy image-processing project. In *Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2016 Fourth International Workshop on*. IEEE, 9–16.

[31] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (2008), 40–53.

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[33] David Pellerin and Scott Thibault. 2005. *Practical FPGA programming in C*. Prentice Hall Press.

[34] Dirk Pflüger, Miriam Mehl, Julian Valentin, Florian Lindner, David Pfander, Stefan Wagner, Daniel Graziotin, and Yang Wang. 2016. The scalability-efficiency/maintainability-portability trade-off in simulation software engineering: Examples and a preliminary systematic literature review. In *Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2016 Fourth International Workshop on*. IEEE, 26–34.

[35] Matthias Richerzhagen. 2016. *Desing and Application of scalable, parallel artifical neural network*. Master's thesis. Aachen University of Applied Sciences. Orig. in German.

[36] Miriam Schmidberger and Bernd Brügge. 2012. Need of Software Engineering Methods for High Performance Computing Applications. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*. IEEE, 40–46.

[37] Martin Schultz. 2015. The TOAR Database of Global Surface Ozone Observations. In *AGU Fall Meeting Abstracts*.

[38] Judith Segal and Chris Morris. 2008. Developing Scientific Software. *IEEE Software* 25, 4 (2008), 18–20.

[39] Hanmao Shi and Jonathan Schaeffer. 1992. Parallel Sorting by Regular Sampling. *Journal of parallel and distributed computing* 14, 4 (1992), 361–372.

[40] Magnus Thorstein Sletholt, Jo Hannay, Dietmar Pfahl, Hans Christian Benestad, and Hans Petter Langtangen. 2011. A literature review of agile practices and their effects in scientific software development. In *Proceedings of the 4th international workshop on software engineering for computational science and engineering*. ACM, 1–9.

[41] SĆ Sonnenburg, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, VojtĂ Franc, et al. 2010. The SHOGUN Machine Learning Toolbox. *Journal of Machine Learning Research* 11 (2010), 1799–1802.

[42] John E Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[43] Devesh Tiwari, Simona Boboila, Sudharshan S Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines.. In *FAST*. 119–132.

[44] James A Whittaker, Jason Arbon, and Jeff Carollo. 2012. *How Google Tests Software*. Addison-Wesley.

[45] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. 2014. Best practices for scientific computing. *PLoS biology* 12, 1 (2014), e1001745.