Scalable Data Analysis in High Performance Computing

Markus Götz

180 ECTS dissertation submitted in partial fulfillment of a
Philosophiae Doctor degree in Computational Engineering

Supervisor
Morris Riedel

Doctoral Committee
Matthias Book
Ólafur Pétur Pálsson
Morris Riedel

Opponents
Håkan Grahn
Shantenu Jha

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavík, December 2017

Scalable Data Analysis in High Performance Computing
Dissertation submitted in partial fulfillment of a *Philosophiae Doctor* degree in Computational Engineering

Copyright © Markus Götz 2017
All rights reserved

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Sæmundargata 2
101, Reykjavík
Iceland

Telephone: +354 525 4000

Bibliographic information:
Markus Götz, 2017, *Scalable Data Analysis in High Performance Computing*,
PhD dissertation, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science,
University of Iceland

ISBN 978-9935-9383-2-9

Printing: Háskólaprent
Reykjavík, Iceland, December 2017

Abstract

Over the last decades one could observe a drastic increase in the generation and storage of data in both, industry and science. While the field of data analysis is not new, it is now facing the challenge of coping with an increasing size, bandwidth and complexity of data. This renders traditional analysis methods and algorithms ineffective. This problem has been coined as the *Big Data* challenge. Concretely in science the major data producers are large-scale monolithic experiments and the outputs of domain simulations. Up until now, most of this data has not yet been completely analyzed, but rather stored in data repositories for later consideration due to the lack of efficient means of processing. As a consequence, there is a need for large-scale data analysis frameworks and algorithm libraries allowing to study these datasets. In context of scientific applications, potentially coupled with legacy simulations, the designated target platform are heterogeneous *high-performance computing* systems.

This thesis proposes a design and prototypical realization of such a framework based on the experience collected from empirical applications. For this, selected scientific use cases, with an emphasis on earth sciences, were studied. In particular, these are object segmentation in point cloud data and biological imagery, outlier detection in oceanographic time-series data as well as land cover type classification in remote sensing images. In order to deal with the data amounts, two analysis algorithms have been parallelized for shared- and distributed-memory systems. Concretely, these are *HPDBSCAN*, a density-based clustering algorithm, as well as *Distributed Max-Trees*, a filtering step for images. The presented parallelization strategies have been abstracted into a generalized paradigm, enabling the formulation of scalable algorithms for other similar analysis methods. Moreover, it permits the definition of requirements for the design of a large-scale data analysis framework and algorithm library for heterogeneous, distributed high-performance computing systems. In line with that, the thesis presents a prototypical realization called *Juelich Machine Learning Library (JuML)*, providing essential low-level components and readily usable analysis algorithm implementations.

Keywords—Data Analysis, Machine Learning, High-Performance Computing, Framework Design, Earth Sciences, Use Case Study

Ágrip

Á síðastliðnum áratug hefur orðið mikil aukning í framleiðslu og geymslu gagna í iðnaði sem og rannsóknum. Þrátt fyrir að gagnagreining sé ekki ný af nálinni, stendur hún frammi fyrir þeirri áskorun að ráða við síaukið magn, bandvídd og flækjustig gagna. Þetta gerir hefðbundnar aðferðir óskilvirkar og hefur þetta vandamál verið nefnt gagnagnótt (e. *Big Data*). Í vísindum koma gögn helst frá umfangsmiklum tilraunum og hermunum. Hingað til hefur ekki verið fyllilega unnið úr gögnunum, heldur hafa þau verið geymd í gagnageymslum fyrir greiningu síðar meir, vegna skorts á skilvirkum úrvinnsluaðferðum. Af þessu má draga þá ályktun að til að greina þessi gögn þurfi víðtæka umgjörð fyrir gagnagreiningu og algrímasöfn og er tölvuumhverfið sem miðað er við, misleit kerfi sem ætluð eru fyrir stórfellda tölvuvinnslu (e. *high-performance computing*).

Þessi ritgerð leggur til hönnun og frumgerðarútfærslu á slíkri umgjörð sem byggir á reynslu sem fengin er úr raunverulegum notkunardæmum, einkum jarðvísindum. Sérstaklega voru skoðuð dæmi um merkingu útlína hluta í punktaskýsgögnum og líffræðilegu myndefni, útlagar (e. outliers) í haffræðilegum tímaraðagögnum og flokkun á fjarkönnunarmyndefni. Til að ráða við hið mikla magn gagna voru tvö greiningaralgrím aðlöguð fyrir samhliða vinnslu í kerfum með samnota- og dreift minni. Þetta eru *HPDBSCAN*, sem er klösunaraðferð byggð á þéttiföllum og *Distributed Max-Trees*, síunaralgrím fyrir myndir. Báðar aðferðir voru færðar yfir í almenna frumgerð sem einfaldar framsetningu skalanlegra algríma fyrir aðrar sambærilegar greiningaraðferðir. Þar að auki gerir þetta kleift að setja fram skilgreiningu á þörfum fyrir hönnun víðtækra gagnagreiningaumgjarðar og söfn algríma fyrir misleit kerfi ætluð til dreifðrar stórtækrar tölvuvinnslu. Að lokum er frumgerð á útfærslu slíkrar umgjarðar kynnt sem nefnd er *Juelich Machine Learning Library (JuML)*, sem veitir aðgang að lágtæknieiningum og tilbúnum útfærslum á greiningaralgrímum.

Lykilorð—Gagnagreining, Gagnagreiningaumgjörð, Reiknigreind, Ofurtölvureikningar, Jarðvísindi, Raundæmi.

List of Publications

Paper I
M. Götz, M. Richerzhagen, C. Bodenstein, G. Cavallaro, P. Glock, M. Riedel, J. A. Benediktsson, “On Scalable Data Mining Techniques for Earth Science”, in the journal *Elsevier Procedia Computer Science*, 51(C), pp. 2188-2197, 2015.

Paper II
M. Götz, C. Bodenstein, M. Riedel, “HPDBSCAN: Highly Parallel DBSCAN”, in *ACM Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, USA, pp. 1-10, 2015.

Paper III
M. Götz, M. Kononets, C. Bodenstein, M. Riedel, M. Book, O. P. Palsson, “Automatic Water Mixing Event Identification in the Koljö Fjord Observatory Data”, **submitted** to *International Journal of Data Science and Analytics.*

Paper IV
C. Bodenstein, **M. Götz**, A. Jansen, H. Scholz, M. Riedel, “Automatic Object Detection Using DBSCAN for Counting Intoxicated Flies in the FLORIDA Assay”, in *Proceedings of 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Los Angeles, USA, pp. 746-751, 2016.

Paper V
M. Götz, G. Cavallaro, T. Géraud, M. Book, M. Riedel, “Parallel Computation of Component Trees on Distributed Memory Machines”, **submitted** to *Transactions on Parallel and Distributed Systems.*

Paper VI
M. Götz, M. Book, C. Bodenstein, M. Riedel, “Supporting Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML Framework”, in *ACM Proceedings of the International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, pp. 1-8, 2017.

Additional Papers

Paper VII

G. Cavallaro, M. Riedel, J. A. Benediktsson, **M. Götz**, T. Runarsson, K. Jonasson, and T. Lippert, "Smart Data Analytics Methods for Remote Sensing Applications", in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Québec, Canada pp. 1405-1408, 2014.

Paper VIII

G. Cavallaro, M. Riedel, **M. Götz**, C. Bodenstein, M. Richerzhagen, P. Glock, and J. A. Benediktsson, "Scalable Developments for Big Data Analytics in Remote Sensing", in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Milan, Italy, pp. 1366-1369, 2015.

Paper IX

M. Riedel, **M. Götz**, M . Richerzhagen, P. Glock, C. Bodenstein, A. S. Memon, M. S. Memon, "Scalable and Parallel Machine Learning Algorithms for Statistical Data Mining—Practice & Experience", in *Proceedings of the IEEE 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, pp. 204-209, 2015.

Software

Software I
M. Götz, C.Bodenstein, "HPDBSCAN", Parallel and scalable C++ implementation of the density-based spatial clustering for applications with noise (DBSCAN) algorithm. Employs the Message-Passing Interface (MPI) and Open Multiprocessing (OpenMP) for the parallelization strategy. Data is processed in the Hierarchical Dataformat 5.
`https://bitbucket.org/markus.goetz/hpdbscan`, 2015.

Software II
Christian Bodenstein, **M. Götz**, "Fly Detector", GUI program allowing the intoxicated fly count analysis in the FLORIDA assay. It is implemented in C++ using Qt and utilizes a shared-memory OpenMP-parallelized flavor of HPDBSCAN for the segmentation step. The code includes the steering software for the FLORIDA assay hardware setup.
`https://github.com/cbodenst/FlyDetector`, 2015.

Software III
M. Götz, "Koljö Fjord Observatory Analysis", Analsys script for the detection of water mixing events in the Koljö fjord observatory time series data. The implementation relies on DBSCAN and confidence interval analysis to detect the events and is implemented in Python. An optional grid search optimizes the model parameters and is parallelized using MPI.
`https://github.com/Markus-Goetz/pangaea`, 2016.

Software IV
M. Götz, G. Cavallaro, "DMT", This C++ code allows the computation of large-scale image max-trees. It is hybrid parallelized using threads and MPI and loads and stores its data in the HDF5 format.
`https://bitbucket.org/markus.goetz/dmt`, 2017.

Software V
M. Götz, C. Bodenstein, M. Richerzhagen, P. Glock, "JuML", The Juelich Machine Learning Library is a C++ and Python-based framework and data analysis algorithm library for high-performance computing systems. It is able to compute on processor and accelerators. The parallelization of the algorithms is realized through CUDA, OpenCL and MPI.
`https://github.com/FZJ-JSC/JuML`, 2016.

List of Figures

1.1 BPMN 2.0 diagram depicting the methodological process of the thesis. . . . 4

2.1 Visualization of the major data analysis tasks. . . . 10
2.2 Common architectures of high-performance computing systems. . . . 13

3.1 Example point cloud from the old-town of Bremen. . . . 18
3.2 Example *Density Based Spatial Clustering for Applications with Noise* (DBSCAN) clustering with $minPoints = 4$. . . . 19
3.3 Spatial domain decomposition strategies for parallelizing DBSCAN. The partition boundary between the processors is shown as a dashed line and the overlapping halo cells in a hatched pattern. 20
3.4 Experimental evaluation of *Highly Parallel DBSCAN* (HPDBSCAN)'s performance. . . . 21
3.5 Schematic of the *Full Loss Of Righting Reflex InDuced by Alcohol* (FLORIDA) assay experiment's hardware setup. . . . 23
3.6 Processing stages of the FLORIDA assay depth-perception problem. 23
3.7 The Koljö fjord observatory. . . . 25
3.8 F1 measure surface for the experimental parameters ε and $minPoints$. . . . 27
3.9 Aerial image of the city of Rome and its land cover types. . . . 28
3.10 Example of max-tree representation based on an exemplary image and its components C_i^c, with the subscript c being the gray-level and the superscript i the canonical point uniquely identifying the component. . . . 30
3.11 Experimental evaluation of the distributed max-tree algorithm. . . . 30

4.1 One-dimensional data decomposition across the samples, including halo zones in purple and duplicates with a hatched pattern. . . . 39
4.2 UML class diagram [1] of *the Juelich Machine Learning Library* (JuML)'s system structure. . . . 42

List of Tables

1.1 Relation matrix of publication contributions and research questions. . 7
1.2 Relation matrix of software contributions and publications. 8

Abbreviations

API Application Programming Interface

ARIMA Auto-Regressive Integrated Moving-Average

BPMN Business Process Modelling Notation

CRISP-DM CRoss Industry Standard Process for Data Mining

CUDA Compute Unified Device Architecture

DAAL Data Analytics Acceleration Library

DBSCAN Density Based Spatial Clustering for Applications with Noise

FLORIDA Full Loss Of Righting Reflex InDuced by Alcohol

FPGA Field Programmable Gate Array

GPGPU General Purpose Graphics Processing Unit

HAF Helmholtz Analytics Framework

HDF5 Hierarchical Data Format 5

HDFS Hadoop Distributed File System

HPC High Performance Computing

HPDBSCAN Highly Parallel DBSCAN

HTC High Throughput Computing

JuML the Juelich Machine Learning Library

KDD Knowledge Discovery in Databases

MIC Many-Integrated-Cores

MIMD Multiple-Instructions-Multiple-Data

MPI Message Passing Interface

NN Neural Network

NUMA Non-Uniform Memory Access

OpenACC Open Accelerators

OpenCL Open Computing Language

OpenHMPP Open Hybrid Multicore Parallel Programming

OpenMP Open Multiprocessing

OS Operating System

RBF Radial Basis Function

RQ Research Question

SDAP Self-Dual Attribute Profile

SEMMA Sample-Explore-Modify-Model-Assess

SIFT Scale-Invariant Feature Transform

SIMD Single-Instruction-Multiple-Data

SKA Square Kilometer Array

SOM Self-Organizing Map

SVM Support Vector Machine

TOS Tree Of Shapes

UMA Uniform Memory Access

Acknowledgments

I am grateful to my supervisor Prof. Morris Riedel, head of the research group *High Productivity Data Processing* at the *Federated Systems and Data* division of the *Juelich Supercomputing Center*, Germany, and *School of Engineering and Natural Sciences* at the *University of Iceland*, who trusted in my abilities and made the doctoral studies possible, as well as for the experiences in last years.

Profound thanks go equally to Prof. Matthias Book and Prof. Ólafur Pétur Pálsson at the *Faculty of Industrial Engineering, Mechanical Engineering and Computer Science* for partaking in my doctoral committee and their guidance on the path to this thesis.

I also would like to thank the Research Center Jülich and the state North-Rhine Westphalia for financially supporting this work.

A special shotout goes to all my collaborators and office mates for an open ear, advise and cooperation. I am looking at you Christian Bodenstein, Philipp Glock, Gabrielle Cavallaro and Mikhail Kononets.

I would also like to thank my friends and Jülich coworkers Paul Baumeister, Salem El-Sayed, Björn Hagemeier, Andreas Herten, Daniel Mallmann, Theodor Nikolov, Lena Oden, Maria Petrova El-Sayed and Rajveer Saini

My good friends Frank and Lysann deserve praise for proof-reading, discussions and taking my mind off at the right times.

I am grateful to my family and especially my mom for believing in me and getting me to the point in my life I am at right now.

Finally, the biggest *'Thank you!'* is addressed to my wife Ewa, for her support, patience, understanding and love, which helped me enormously in finishing this undertaking and for being at my side no matter my spirit.

In memory of Vitali and Maria.

Contents

List of Publications . . . v

List of Figures . . . ix

List of Tables . . . xii

Abbreviations . . . xiv

Acknowledgments . . . xvi

1 Introduction . . . 1
- 1.1 Motivation . . . 1
- 1.2 Research Questions . . . 3
- 1.3 Outline . . . 5
  - 1.3.1 Covering Paper . . . 5
  - 1.3.2 Appended Papers . . . 6
- 1.4 Contributions . . . 7

2 Background . . . 9
- 2.1 Data Analysis . . . 9
  - 2.1.1 Learning Approaches . . . 9
  - 2.1.2 Learning Tasks . . . 10
  - 2.1.3 Data Analysis Process . . . 11
- 2.2 High-Performance Computing . . . 12
  - 2.2.1 System Architectures . . . 12
  - 2.2.2 Programming Models . . . 14

3 Summary of the Publications . . . 17
- 3.1 Paper I . . . 17
- 3.2 Paper II . . . 19
  - 3.2.1 DBSCAN . . . 19
  - 3.2.2 Parallelization Strategy . . . 20
- 3.3 Paper III . . . 22
  - 3.3.1 The FLORIDA Assay . . . 22
  - 3.3.2 Depth-Separated Image Segmentation . . . 22

3.4 Paper IV 25
3.4.1 Koljö Fjord Observatory 25
3.4.2 Water Mixing Event Detection 26
3.5 Paper V 28
3.5.1 Land Cover Type Classification 28
3.5.2 Feature Engineering and Component Trees 29
3.5.3 Parallelization Strategy 30
3.6 Paper VI 32

4 An HPC Data Analysis Framework 35
4.1 Related Work 36
4.2 Data Analysis Algorithm Parallelization Paradigm 38
4.3 Requirements 40
4.4 Juelich Machine Learning Library 41

5 Conclusions 45
5.1 Future Work 47

References 49

Appendices 58

A Paper I 61

B Paper II 73

C Paper III 85

D Paper IV 93

E Paper V 109

F Paper VI 127

Chapter 1

Introduction

1.1 Motivation

In the last decades business and industry have seen an influx of data-oriented and data-intensive services [2]. Search engines [3], social networks, business intelligence [4] and the Internet of Things are just some of the examples for this trend. With an even earlier starting point, science has undergone a process very much alike. Large-scale experiments—exemplified by the TOAR weather and climate research database [5], PANGAEA, an environmental science data collection [6], or the planned radio observatory *Square Kilometer Array* (SKA) [7]—collect an exponentially growing amount of data in need of analysis. To provide a convenient term for these phenomena, the start of the Big Data era has been announced. This term generally refers to data processing and analysis problems, which cannot be adequately dealt with using conventional technology. Initially, the focus has been put on the so-called three *V's* [8], i.e. keywords starting with the letter *V*, attempting to deliver a concise description of the involved challenges. These are *volume*, *velocity* and *variety* and correspond to the quantity, bandwidth as well as number of different sources and types of data involved in an analysis problem. Later, additional V's have been added to better reflect challenges previously unaccounted for, starting at four [9], to seven [10], then ten [11] and ultimately a not quite genuine 42 [12]. They describe related issues, such as how to visualize the data, price them, make data secure against theft or alteration and so forth.

In line with the inevitable expansion of Big Data challenges, technologies attempting to overcome them have seen accelerated research and investment. Particularly in the industry, the concepts of *High Throughput Computing* (HTC) and *cloud computing* have established themselves as cost-efficient and scalable ways of processing data, while also ensuring agreed-upon service levels in terms of availability, redundancy of computation, etc. This infrastructure-centered pursuit has in turn given rise to new technology stacks, such as the *Hadoop Distributed File System* (HDFS) [13], parallel processing platforms like Hadoop [14] or Spark [15], virtualization solutions including CloudStack and Docker, and database management systems like NoSQL or in-memory databases. Most of them, however, have strong

limitations. Computational processing power in HTC is constrained by the commodity hardware it is run on, along with simple, commercial-grade interconnects and as a result, limited communication bandwidth and basic topological structure. While this is sufficient for industrial applications, mainly focused on simple correlated pattern detection in user data, for example, this effectively limits the potential of implementing highly complex, though analytically efficient and accurate models, which are prevalent in scientific research.

Next to traditional simulations, computational data analysis becomes an increasingly more frequent application scenario in science. It is used to perform simulation observation data assimilation, augmentation of experimental data and to process results of measurements gathered by sensor systems. However, despite the frequent similarity of high-level analysis tasks compared to the industry, scientific technological implementation aspects differ heavily. Due to precision constraints, such application scenarios require much more complex and computationally intensive analysis models. This translates not only to immense computational requirements and space consumption, but also the need for expertise and efficiency in employing the resources offered by the traditionally used *High Performance Computing* (HPC) systems. These systems often employ expensive, however computationally powerful heterogeneous hardware components, which are tightly integrated via high-bandwidth interconnects.

A large base of experience and knowledge in computationally aided natural science research comes with decades worth of legacy systems that are costly to change. This results in a certain rigidity against attempts at adapting the dynamically developing HTC technologies, from which the sciences would undoubtedly stand to gain. The technology stacks employed in HPC are largely incompatible with new technologies introduced to the HTC area. Their disjunction starts at low levels with the concrete distributed file systems and follows through data formats, schedulers, programming languages and the possibility to exploit heterogeneous computational hardware. As a result, attempts at merging them would entail substantial engineering efforts. Rather than striving to make them perform as interchangeable equivalents, the arguably better approach is to identify the core aspects of the rising HTC technologies and to apply and adept them to HPC scenarios.

Given the increased demand for powerful data analysis in science and the explained state of technology, there is an objective need for the design and implementation of parallel and scalable solutions. The common use of legacy HPC tools and increasing need for innovation calls for attention from the computational engineering community. To answer this call, this dissertation studies and analyzes a number of different natural science application domains in order to identify novel approaches that are tailored to HPC systems. From the requirements exposed in the studies a generalized parallelization paradigm for HPC data analysis algorithms is derived. It is further implemented in a cohesive prototypical data analysis framework, enabling large-scale data analysis and algorithm development for modern, heterogeneous HPC systems.

1.2 Research Questions

This thesis examines the need for HPC-based data analysis in scientific research and proposes a systematic manner of addressing it. Studies of use cases stemming from a number of Earth sciences enable an in-depth investigation of specific requirements imposed by particular research problems. They further result in the identification of commonalities in the demand for powerful data analysis solutions present across domains. Means of addressing the observed research gap are explored based on scientific literature documenting designs and applications of data analysis algorithms. Here, performance is of primary interest, both in terms of computational efficiency, as well as the extent in which it succeeds to address the domain-specific challenge. The goal is to identify readily available efficient HPC tools applicable to the examined Earth science problems, or lack thereof. If absent, conventional data analysis algorithms that are best suited for the scientific use cases are selected and redesigned to maximize computational performance and optimize the use of HPC resources. Their novel implementations are applied, validated and evaluated in terms of domain-specific insights and computational performance. With the intention to shrink the mentioned gap, due to the scarce availability of HPC data analysis tools, a framework of such ready-to-use algorithms is proposed. A prototypical candidate, including the collection of pre-implemented algorithms, is published open-source.

In this manner the author addresses the following set of research questions, which interweave the conducted studies:

***Research Question* (RQ) 1:** What is the state of the art in data analysis and its technologies in the Earth sciences?

RQ 2: How and which parallel and scalable algorithms can support the analysis of selected Earth science use cases?

RQ 3: Can the identified techniques be applied in a generalized fashion in scientific domains outside of the Earth sciences?

RQ 4: Are there enough commonalities that would justify the design of a parallel and scalable data analysis framework and algorithm library for HPC computing systems?

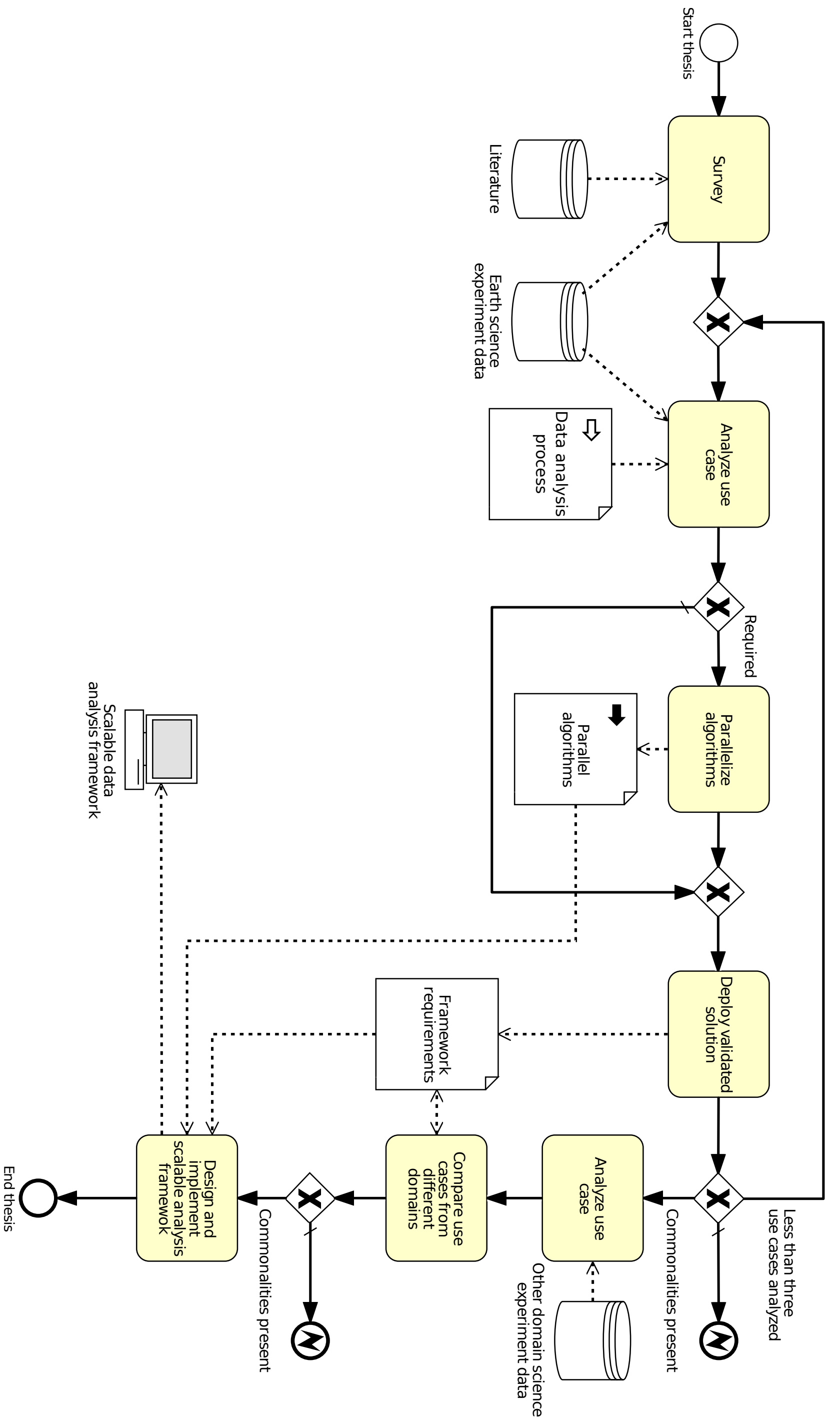


Figure 1.1: BPMN 2.0 diagram depicting the methodological process of the thesis.

Figure 1.1 depicts the method selected for answering these questions in form of a *Business Process Modelling Notation* (BPMN) [16] diagram. It closely follows the posed research questions and tries to answer them gradually. First, the literature and the selected use cases are surveyed, including proposed analysis approached in order to determine the state of the art. Then, the use cases are tackled in conjunction with the experts of the particular application domain. The Earth sciences have been selected as the field of choice due to availability, size and characteristics of publicly available data sets. Depending on the chosen analysis approach, the parallelization of the used algorithms may be necessary in order to be able to handle the data volume and bandwidth present in the use case. The proposed solution should be so generic that it can also be applied to data sets with similar analysis tasks and scalable enough to handle increased data quantities. This is what Research Question three aims at—the ability to generalize the findings. Given that there are enough commonalities, it is sensible to extract these and use them as a requirements list and potentially paradigm for the design of a parallel and scalable framework and data analysis library on HPC systems. This software artifact should support two major user groups: the developers of data analysis algorithms and the actual data analysts. The library should aid the former through the abstraction from underlying hardware features, if possible, while also providing recurring functionality for the implementation of parallel algorithms. In contrast to that, the second group needs a set of reliable and efficient standard algorithms, which require a high-level *Application Programming Interface* (API) working on abstract entities such as entire data sets and thereby handling the parallelization internally—instead of explicit low-level components such as passed messages.

1.3 Outline

This thesis is composed in a cumulative style. The major findings are therefore presented in form of peer-reviewed conference and journal publications as well as pending submissions, which are to be found in the appendix. Publications to which the thesis author only contributed to a lesser extent are deliberately excluded. For a complete list of all publications, however, please refer to the “List of Publications”.

1.3.1 Covering Paper

The following chapters will provide a brief summary of the results as follows.

Chapter 1—Introduction presents the motivation and research methodology for answering the research questions posed in this thesis.

Chapter 2—Background provides background on both, data analysis tasks, methods and the involved process, as well as a background on HPC.

Chapter 3—Summary of the Publications summarizes the appended papers and puts them into context of the posed research questions.

Chapter 4—An HPC Data Analysis Framework proposes a data analysis framework and algorithm library targeting heterogeneous, distributed high-performance computing systems. For this, related work in the field is introduced first, before this thesis' design concept is presented. Along with this, a generalized algorithm parallelization strategy paradigm, abstracted from the publications in Chapter 3, is suggested. It adds to the definition of a requirements list of essential components of the aforementioned framework. Finally, this chapter is concluded with the introduction of *JuML*—this prototypical realization of a large-scale data analysis framework for HPC systems proposed in line with this thesis.

Chapter 5—Conclusions concludes the thesis and presents perspectives for future research opportunities.

1.3.2 Appended Papers

The following papers from the List of Publications can be found in the appendix.

Paper I
M. Götz, M. Richerzhagen, C. Bodenstein, G. Cavallaro, P. Glock, M. Riedel, Morris, J. A. Benediktsson, "On Scalable Data Mining Techniques for Earth Science", in *Elsevier Procedia Computer Science*, 2015, pp. 2188-2197.

Paper II
M. Götz, C. Bodenstein, M. Riedel, "HPDBSCAN: Highly Parallel DBSCAN", in *ACM Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, USA, 2015, pp. 2:1-2:10.

Paper III
C. Bodenstein, **M Götz**, A. Jansen, H. Scholz, M. Riedel, "Automatic Object Detection Using DBSCAN for Counting Intoxicated Flies in the FLORIDA Assay", in *Proceedings of 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Los Angeles, USA, 2016, pp. 746-751.

Paper IV
M. Götz, M. Kononets, C. Bodenstein, M. Riedel, M. Book, O. P. Pálsson, "Automatic Water Mixing Event Identification in the Koljö Fjord Observatory Data", submitted to *International Journal of Data Science and Analytics.*

Paper V
M. Götz, G. Cavallaro, T. Géraud, M. Book, M. Riedel, "Parallel Computation of Component Trees on Distributed Memory Machines", submitted to *Transactions on Parallel and Distributed Systems.*

Paper VI
M Götz, M. Book, C. Bodenstein, M. Riedel, "Supporting Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML Framework", in *ACM Proceedings of the International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2017, pp. 1-8.

1.4 Contributions

The main contributions of this thesis are subdivided into three major categories. First, the analysis of data-intensive, scientific use cases with an emphasis on Earth sciences. The analysis has contributed to or established the state of the art for the particular analysis problem, e.g.:

- unsupervised object segmentation in point clouds (**Paper II**),
- unsupervised depth-separated object segmentation (**Paper III**),
- semi-supervised classification of water mixing events (**Paper IV**) and
- supervised land cover type classification using attribute filters (**Paper V**).

Second, the parallelization of existing data analysis algorithms that are employed in the use cases, targeting distributed-memory environments and thereby enabling scalability with respect to processing time and memory consumption. In particular, these are:

- the formulation of a parallel DBSCAN (**Paper II**) and
- a parallel min- and max-tree algorithm (**Paper V**).

Both have outperformed the state of the art—even by orders of magnitudes with respect to memory consumption.

Third, a generalized paradigm for the design of parallel and scalable data analysis algorithms suitable for processing a large number of samples has been proposed. Following well-established engineering practices, the algorithmic components have been abstracted and bundled into a prototypical data analysis library, called JuML, aimed for the use in heterogeneous distributed-memory HPC systems, which is presented in **Paper VI**. An overview of the relation between the contributions and the publications can be found in Table 1.1.

Table 1.1: Relation matrix of publication contributions and research questions.

RQ \ Paper	Paper I	Paper II	Paper III	Paper IV	Paper V	Paper VI
RQ 1	X					
RQ 2	X	X		X	X	
RQ 3			X			
RQ 4	X	X	X	X	X	X

In line with the thesis, a number of software artifacts have been created, along with JuML, which has been open-sourced. Table 1.2 shows how the software creation and utilization relates to the publications. These are already in productive use. A non-exhaustive list includes the FLORIDA assay software, HPDBSCAN on the Blacklight and DEEP-EST supercomputing systems and the distributed max-tree algorithm at the Research Centre Jülich. JuML will serve as the conceptual template for the planned *Helmholtz Analytics Framework* (HAF).

Table 1.2: Relation matrix of software contributions and publications.

Paper / RQ	Paper I	Paper II	Paper III	Paper IV	Paper V	Paper VI
Software I - HPDBSCAN	X	X	X	X		
Software II - FLORIDA				X		
Software III - PANGAEA			X			
Software IV - DMT					X	
Software V - JuML						X

Chapter 2

Background

2.1 Data Analysis

In this section the methods and processes for analyzing data is presented in detail in order to empower the reader to better follow the use cases introduced in Chapter 3. As a matter of fact, the field is so large that only a small portion of it can be represented here. Therefore, the focus is put on data mining approaches, in particular clustering, as well as machine learning approaches, which have directly or indirectly been used in the problems' analysis. Machine learning enables the purely data-driven construction of systems without explicit programming. This means that the inner workings, or in abstract terms the system's function, is purely derived from patterns within explanatory data. The process of deriving this function is called training or learning. Generally, learners can be distinguished into a number of categories based the way they learn and the data analysis task. The following sections will introduce both concepts briefly.

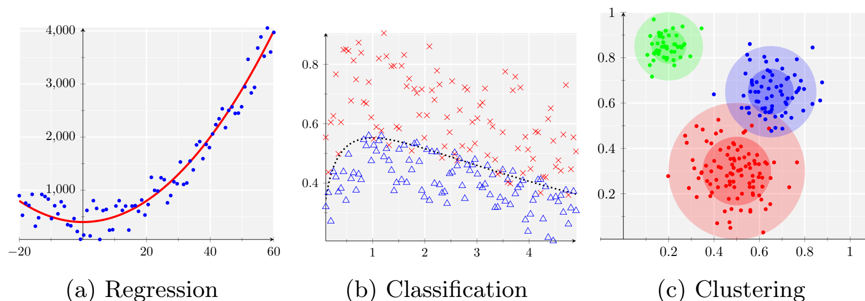
2.1.1 Learning Approaches

Supervised learning occurs when the pattern, or in jargon *ground truth*, is known and provided as additional training input. That is, the learner is trained based on data samples containing both the attributes, i.e. a vector of descriptive values, and the labels, i.e. the pattern. A graphic example from biology could be a system that divides flowers into different species (the label) based on their color, petal sizes and height (the attribute vector).

Unsupervised learning is performed without any ground truth solely on the attributes of the data samples. This is usually employed for explorative data analysis and machine learning and results in the grouping or clustering of the data. In order to be able to tell whether a number of samples belong into one category, their resemblance is judged based on a similarity function, which is maximized during the learning. Reusing the flower example mentioned above, each of the species would ideally be part of a separate group after an unsupervised training.

The last category is *reinforcement learning*. It also does not require any labels. However, in contrast to unsupervised learning, the system would not consider single instances, but the problem as a whole. A reinforcement learner would simply propose a solution, initially often a random one, that would be assessed using a punishment-and-reward-function. Based on the received feedback, the reinforcement learner will memorize good behavior, i.e. the correct parts of the answer, and forget incorrect labeling.

2.1.2 Learning Tasks



(a) Regression (b) Classification (c) Clustering

Figure 2.1: Visualization of the major data analysis tasks.

The second major distinction between different machine learning systems can be made based on the desired output. The most general form of a learning tasks is the so-called *regression* and refers to learning an arbitrarily shaped function. In line with that, the learning system shall be able to determine for any given domain value x the respective codomain value Y. In practice this will usually be an approximation, or prediction, and is denoted as $\hat{Y}$. Typically regression is used in forecasting problems, e.g. stock market or weather prediction. Common models for performing regression are ordinary least squares [17], logistic regression [18], support vector regression [19] and artificial *Neural Network* (NN) [20]. For time series regression there are additional models such as *Auto-Regressive Integrated Moving-Average* (ARIMA) [21] or recurrent neural networks [22].

Classification is a special form of regression problem, in which a function, often piecewise, has to be found that separates classes of samples from each other, usually based on the provided labels. The simplest form of classification is binary, where only two classes exist—i.e. belongs to a given class or not. This basic classifier can then be extended to perform multiclass classification [23]. There are different approaches to do so, such as one-versus-one or one-versus-all for example. They are founded in building binary classifiers that distinguish one particular against all other available, but differ in the way the victor is selected. Common classification models are decision trees [24], random forests [25] (decision tree ensembles), *Support Vector Machine* (SVM) [26] and NN.

Outlier or anomaly detection is a specific form of classification, in which exceptional samples very different or distant from the remainder of the data are to be

found. Such a problem can be solved using supervised classification as introduced above, but can also be identified based on the outliers' dissimilarities towards the other samples. This leads to the third major algorithm group, *clustering*, which is about identifying groups by maximizing the intra-group similarity and the inter-group dissimilarity. Well-known clustering algorithms are hierarchical clustering, k-means [27], DBSCAN [28] and *Self-Organizing Map* (SOM) [29]. Figure 2.1 depicts all of the introduced learning approaches and tasks graphically.

2.1.3 Data Analysis Process

There are a number of standard processes for approaching data-driven analysis problems. The three major ones are *Knowledge Discovery in Databases* (KDD) [30], *Sample-Explore-Modify-Model-Assess* (SEMMA) [31] and *CRoss Industry Standard Process for Data Mining* (CRISP-DM) [32]. Despite small differences, all of these include the same conceptual steps shown below [33]. Based on these steps, one is able to identify the opportunities assisting the analysis process through technical means, from which requirements for the data analysis framework can be derived.

Selection is the process of understanding the domain problem, the data aspects and formats involved in the data analysis challenge. Based on this a suitable dataset needs to be obtained through collection or selection of already existing samples.

Exploration is initially required to understand the properties of the data. For this visualization of the data is necessary as well as the computation of descriptive statistical values. Especially data quality problems need to be uncovered here, such as noise or missing values. Based on that a first model hypothesis can be defined.

Preprocessing is a technical step in which the data quality is improved to the point it can be utilized for model construction. This involves, among others, data interpolation, outlier removal and noise reduction, duplicate elimination, feature engineering, data augmentation and class imbalance correction through synthetization. In practice, data preprocessing is an iterative approach that is revisited multiple times through the entire analysis.

Analysis is the step of the actual model construction. Due to the identified analysis task an appropriate supervised or unsupervised data analysis algorithm is selected and trained to approximate the problem as good as possible. Similar to data preprocessing, this is an iterative step, fine-tuning and optimizing the devised stochastic model.

Evaluation is the step of assessing the model's performance based on some measure, like the prediction accuracy for example, compared to the desired analysis task. Based on that, new models or variants of it need to be considered. An essential part of the evaluation is the fine-tuning of hyperparameters as well as the estimation of its generalization.

Deployment is the phase of setting up the selected model for productive use. This is usually a technical process involving the installation of suitable execution environments, development efforts to scale the analysis pipeline to production data sizes and continuous administration and maintenance.

2.2 High-Performance Computing

In this section a background in the field of HPC is introduced. The main focus is put on explaining the technical aspects of modern clusters and supercomputers to explain why parallel and scalable data analysis algorithms have to fulfill certain technical requirements to make them usable and efficient on these systems. Therefore Subsection 2.2.1 introduces hardware related topics, while Subsection 2.2.2 is presenting programming models available for the implementation of algorithms on top of the explained computer systems.

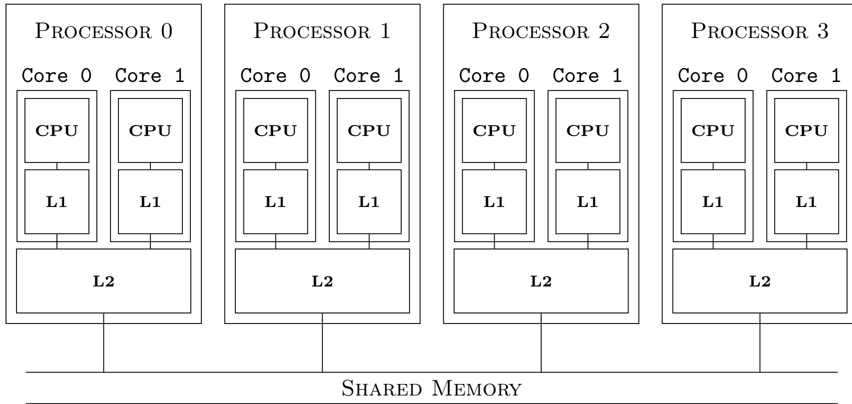
2.2.1 System Architectures

Modern HPC systems and supercomputers are a potpourri of different designs, architectures and hardware platforms. However, some of these are more wide-spread and dominant than others. In particular, these are usually inherently parallel, heterogeneous, modularised commodity cluster systems with multi-core processors and optional accelerators. Figure 2.2 depicts these common architectures or their parts schematically.

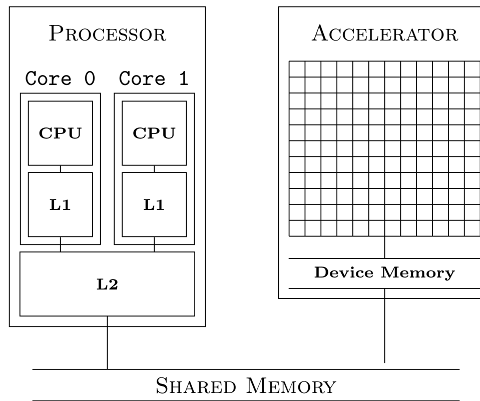
The most basic component of a cluster is a so-called *node*. This is a singular multi-core or -processor computer as can be seen in Figure 2.2a. Each node has a number of processors, which in turn have several cores, representing the most basic computation unit. Cores can exchange information with each other via an on-chip network and a shared memory accessible via a hierarchy of intermediary, faster to access caches [1]. An arbitrary memory entry can potentially reside in multiple caches of different cores at once. In order to prevent data races, it must be kept consistent across all cores, realized by a so-called *cache-coherence-protocol*. Finally, the shared memory is nowadays often divided into parts that can be either slower or faster accessed by a given core or processor. Such an approach is called *Non-Uniform Memory Access* (NUMA), opposed to the traditional *Uniform Memory Access* (UMA). Processors in shared memory architectures usually work according to the *Multiple-Instructions-Multiple-Data* (MIMD) principle [34]. This means that each processor can execute a different program on different parts of the memory at any given time, consequently allowing efficient execution of task-parallel workloads.

Also, there is a trend in the last decade to utilize special accelerator hardware. Well-known examples are *Field Programmable Gate Array* (FPGA), *Many-Integrated-Cores* (MIC) or *General Purpose Graphics Processing Unit* (GPGPU) technologies. These are usually designed (especially the latter) to work in a *Single-Instruction-Multiple-Data* (SIMD) [34] fashion. This means that each of the cores on the accelerator execute the same operation at the same time, often in a lockstep

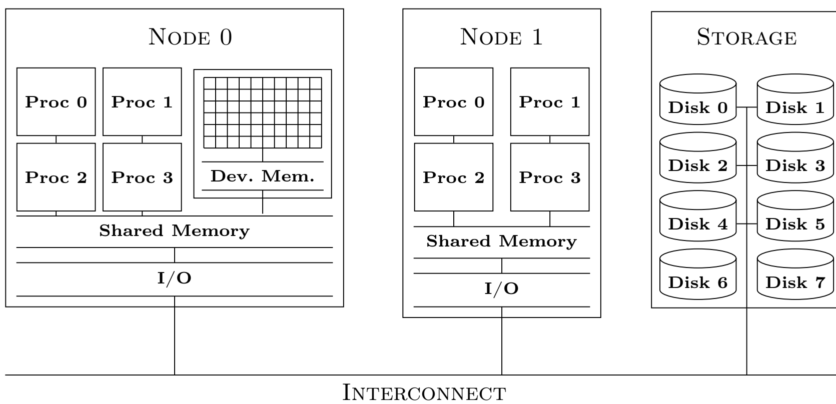
[1] Usually denoted by a capital L for the level and a number for the hierarchy distance to the core.



(a) Shared-memory.



(b) Accelerators.



(c) Distributed-memory.

Figure 2.2: Common architectures of high-performance computing systems.

mode. In line with that, independent and embarrassingly parallel vector operations, such as for example stencils or tensor operators, are the preferred computational problem to be solved using accelerators. Fitting for this use case, the number of processing units or cores is usually very high, compared to the previously introduced processor-based shared memory design, with core counts ranging from dozens to hundreds or thousands (depicted as grid in Figure 2.2b), though they are individually less versatile and fast. An accelerator is usually fed data from the shared memory via the bus of the so-called *host processor* and has to be explicitly transferred between the two devices. This adds significant overhead due to bus and memory bandwidth limitations, if repeatedly done, and should be avoided in favor of more computations on the accelerator. Distinct accelerator applications in data-analysis include, among others, computation of sample statistics, preprocessing and normalization of the datasets and matrix and tensor multiplications in neural networks.

The scalability of shared memory designs or accelerators has inherent boundaries. This can be attributed, for instance, to overheads in synchronization, physical heat development on the dies, signal conductor delays or, particularly for accelerators, unfitting computational problems. Therefore, in order to further increase computational processing performance and capabilities, one needs to scale out horizontally. One can usually achieve this in most modern HPC and supercomputer systems by connecting multiple nodes using an interconnect with one another into a computer cluster system. Due to the separate memories this is also called a *distributed memory system* (a schematic depiction is displayed in Figure 2.2c). In the past the individual nodes used to be homogeneous, i.e., of the same kind. However, nowadays systems tend to be heterogeneous, meaning nodes with different capabilities are mixed within the system. For example, there could be nodes with more or less local memory, different processor counts or additional built-in accelerators. The storage system, or simply the hard disk, can be either node-local or, especially for larger systems, a separate entity that is attached to the interconnect. This allows the independent scaling of the I/O system and enables parallel data access.

2.2.2 Programming Models

The de-facto standard for programming distributed memory systems in the context of HPC is the *Message Passing Interface* (MPI) [35]. It is an API with over 500 functions that define a set of communication primitives to pass binary data. They can be roughly divided into two categories.

Firstly, there are point-to-point message passing operations between particular nodes, such as send and receive. Both the sender and recipient are uniquely identified by a contiguously enumerated ID for a node, called *rank*, in the interval $[0, size[$, with *size* being the total number of nodes. This information is encapsulated into handle for the nodes set that is called a *communicator*.

Secondly, there are collective operations. These are many-to-one or many-to-many operations that can be implemented efficiently across larger node counts, using communication trees for example, and are the major reason for MPI's success in HPC. Examples for collective operations are reduction operations, data

broadcasts or gathering. The actual low-level networking library code needs to be provided by the implenter of the particular MPI stack and needs to be tailored to the used interconnect. Prominent interconnect representatives are for example MyriNet [36], Infiniband [37] or Omni-Path [38].

Parallel programming models for shared memory architectures are slightly more diverse, but have matured to a point that only a small number of major ones exist. First and foremost, there is traditional thread-based parallelization. Being an *Operating System* (OS) resource concept, each of them provide their own API for creating and managing threads. Higher-level languages, such as Python, and as of late also traditional languages such as C++, offer OS-independent abstractions. The typical parallelization approach is to spawn a single thread per core and assign it a part or sub-task of the problem, e.g. chunks of an input vector or tensor an operation needs to be applied to.

Due to the fact that this pattern is so common, another, annotation-based programming model called *Open Multiprocessing* (OpenMP) [39] has established itself. It encapsulates these patterns usually in a few lines of code annotations, which are in turn translated by the supporting compiler into parallel code, mostly based on threads that are spawned from the main or *master* execution thread. Moreover, OpenMP also assists with the synchronization by providing atomic variables, barriers and so forth. A combination of the OpenMP and MPI programming models in a single application is often referred to as a *hybrid*.

Lately, OpenMP has also adopted an extension that allows to program accelerators. This adds to the large potpourri of programming models and languages designed for these specialized architectures. Alternatives for annotation-based parallelization with OpenMP are *Open Hybrid Multicore Parallel Programming* (OpenHMPP) [40] and *Open Accelerators* (OpenACC) [41], following the same API design principles. An programmatic alternative to that is *Open Computing Language* (OpenCL) [42], which allows the implementation of more fine granular parallelization strategies. In principle it can be used for any kind of acceleration hardware, given that a supporting compiler exists, and is in practice often used for GPGPUs. This makes OpenCL the major alternative to Nvidia's proprietary *Compute Unified Device Architecture* (CUDA) GPGPU [43] programming model. The latter is highly optimized for, but also restricted to the companies' hardware. It enjoys widespread use due to its early release, the widespread of Nvidia GPGPUs in HPC and its lean API, compared to OpenCL, because of the narrower focus. However, a number of vendors, e.g. Intel and Altera, have picked up OpenCL as a standard programming model for their MICs [44] and FPGAs respectively, making it highly versatile.

It should have become clear that in order to efficiently exploit the capabilities of modern HPC systems, the development of hybrid applications, employing multiple programming models at once, is necessary. As a result the development time and complexity increases, especially with respect to portability to new HPC systems and technologies. Therefore, a number of wrapper technologies and libraries start to appear. The two major ones being ArrayFire [45] and TensorFlow [46]. These do not only encapsulate the complexity of the different programming models and automatically select the right operation kernel depending on the selected execution device, but are also more abstract in terms of the offered API. This means they include already parallelized, high-level vector operations such as statistics, stencil filters or arithmetical operators. They can be seen as more feature-complete basic linear algebra libraries, which also offers a number of highly optimized routines needed in data analysis. While they are a solid foundation to develop on single heterogeneous nodes, one needs to also take into account that they cannot efficiently exploit distributed computing resources and in line with that the capabilities of modern HPC systems. Therefore, a truly large-scale data analysis framework in the context of HPC needs to push the boundaries further and also utilize MPI to allow stronger, horizontal scaling. As stated in RQ four, one major task is to design and develop such a toolkit, if possible. Therefore, it follows that it also needs to utilize the presented parallel programming models.

Chapter 3

Summary of the Publications

3.1 Paper I: On Scalable Data Mining Techniques for Earth Science

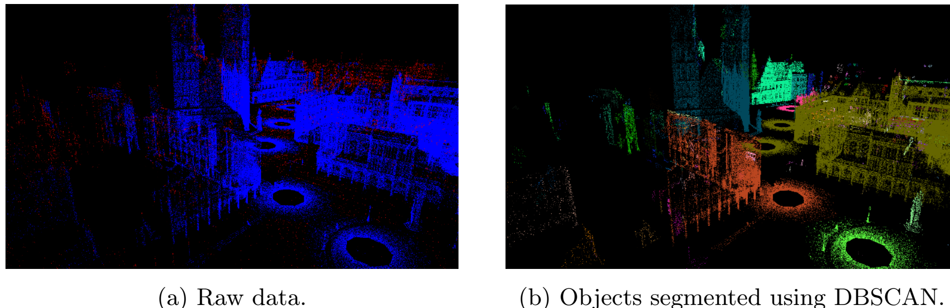
M. Götz, M. Richerzhagen, C. Bodenstein, G. Cavallaro, P. Glock, M. Riedel, Morris, J. A. Benediktsson, "On Scalable Data Mining Techniques for Earth Science", in the journal *Elsevier Procedia Computer Science*, 51(C), pp. 2188-2197, 2015.

This publication contributes to the first research question by surveying the current state of the art in parallel libraries and tooling for data analysis with a focus on Earth Sciences.

This paper presents a technology survey of non-commercial, open-source tools and libraries for two different data analysis algorithms: non-linear SVMs and DBSCAN. In line with that, their suitability to handle current and future Big Data datasets have been evaluated (contribution to **RQ 1**). This includes, among other things, the possibility to deploy the implementations on HPC systems. Hence, a number of technical capabilities have been evaluated, such as the inclusion of parallel I/O, the degree of parallelization and the correctness of the computed results.

The first part of the publication is concerned with parallel SVMs [26]. Only three out of the twelve evaluated candidate tools have parallel implementations to begin with and were therefore investigated futher. The Apache Spark-based implementation in MLlib [47] is one of them. It is severely limited in analysis capability, due to the fact that it only supports linear kernel functions. This means that classification problems with a non-linearly separable decision boundary, i.e. interlaced instances of different classes, cannot be solved. As a result, MLlib is unable to construct effective classification models for typical real-world applications that are especially predominant in science. The other two candidates are GPU LibSVM [48], which only supports CUDA devices, and an MPI-based solution called πSVM [49], which is targeting CPUs. Both of them posses the possibility to construct non-linear models through the implementation of advanced kernel functions.

Based on its dissemination, the latter has been selected for further optimization. In particular, the memory access and communication pattern—and through that workload balancing—were sub-optimal. In the publication an improved access strategy is proposed, so that the number of computations is equal on each of the processing nodes. A detailed explanation of this strategy and an implementation of this concept is given in the bachelor thesis of Matthias Richerzhagen Richerzhagen [50], supervised by the author.



(a) Raw data.

(b) Objects segmented using DBSCAN.

Figure 3.1: Example point cloud from the old-town of Bremen.

The second set of surveyed implementation concerns the unsupervised clustering algorithm DBSCAN (see also Section 3.2.1), motivated by an object segmentation use case in multi-dimensional point clouds. Point clouds are point-wise recordings of spatial locations in a given coordinate space, usually the euclidean. They are often used to represent objects or surfaces like landmasses or areas. The recorded values include the spatial three-dimensional vector components x, y and z, as well as potentially additional values such as parts of the electro-magnetic spectrum, like heat radiation, for example. Analysis of the data usually try to reconstruct and segment objects present in reality [51]. Figure 3.1 depicts a point cloud of the old town of Bremen, Germany, captured using an aerial drone scan as well as the result of a clustering-based object segmentation of the buildings. The survey shows that only a limit set of data analysis libraries support a DBSCAN-based analysis to begin with, and if so, are not well parallelized. In particular for HPC technology stacks, there is only one implementation available called PDSDBSCAN [52], which has fundamental flaws in the utilization of memory. This was the motivation to review the parallelization strategy, resulting in HPDBSCAN presented in Section 3.2.

The publication concludes that the current situation in availability and stability of scalable data analysis tools needs improvement. Based on the two presented algorithm examples, which are both in widespread use in applications, it is argued that the ability to process large data sets is limited. Even though data analysis is, in terms of resource utilization, still far behind traditional HPC applications, like numerical simulations, one can expect an increasing demand. Therefore, scalable and parallel algorithm implementation are required and engineering and research efforts should be increased in this field.

3.2 Paper II: HPDBSCAN—Highly Parallel DBSCAN

M. Götz, C. Bodenstein, M. Riedel, "HPDBSCAN: Highly Parallel DBSCAN", in *ACM Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, USA, pp. 2:1-2:10, 2015.

This publication contributes to the second research questions as it explains a scalable parallelization strategy for DBSCAN, which can be used, for example, for point cloud analysis in the Earth Sciences. Moreover, it contributes to the fourth research question as it utilizes recurring components, which will be explained later in Section 3.5 and Chapter 4, that can be abstracted into a generalized analysis framework.

The following section introduces the original DBSCAN clustering algorithm first. Afterwards, Section 3.2.2 presents the parallelization strategy envisioned in the context of this thesis in details.

3.2.1 DBSCAN

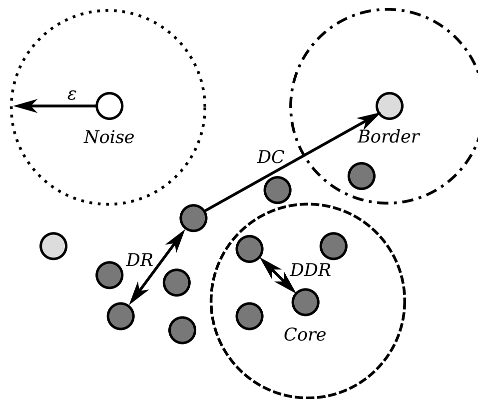


Figure 3.2: Example DBSCAN clustering with $minPoints = 4$.

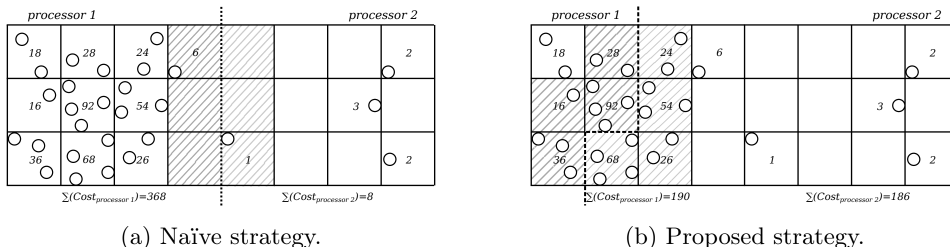
In 1996 Ester et al. [28] introduced a novel density-based clustering algorithm group by formulating DBSCAN. Its main principle is to recursively expand clusters through the evaluation of a spatial density criteria. For this, it scans the entire database of points and evaluates for each of them in a search radius ε how many neighboring points exist using a given distance function *dist*. If the density criteria *minPoints* for the number of neighbors is satisfied, then the current point is a core point of a cluster. This cluster may be an entirely new set or is absorbed by the cluster of a neighboring core point within the search radius. If a point is neither absorbed nor fulfills the core criteria itself, it is marked as noise. DBSCAN has a number of advantageous properties compared to other clustering algorithms,

such as k-means [27]. First, it has a lower number of intuitively selectable parameters, which can often be easily determined for lower-dimensional problems. Second, DBSCAN is able to identify a previously unknown number of arbitrarily shaped clusters. This is a powerful analysis property as it reduces the number of potentially hard-to-determine model parameters. Finally, the algorithm is robust towards noise due to a built-in filtering mechanism, enabling the analysis of datasets with strong outliers. Especially the last property is beneficial for actual real-world analysis as the use cases in Section 3.3 and 3.4 are going to demonstrate. Figure 3.2 depicts an exemplary DBSCAN clustering.

3.2.2 Parallelization Strategy

Paper II presents a parallelization strategy for DBSCAN called HPDBSCAN (contributing to **RQ 2**). Its core concept follows a divide-and-conquer-approach: the entire dataset is initially equally divided among all available processing cores, then a local partial DBSCAN result is computed, which can then successively merged into a correct global clustering.

In skewed datasets, the spatial partition should not be carried out in regular intervals, because the point density of the spatial chunks may be highly imbalanced and result in an inferior workload-balance. Therefore, the publication proposes a split heuristic based on the number of compute-intensive evaluations of the *dist* function. For this, each point is uniquely associated with an ε-sized spatial cell of an overlayed, disjoint hypergrid index structure. Each cell receives a score—the sum of the number of points within the cell and its direct neighbors, equal to the *dist* invocations—which is instead evenly divided among the available processing cores. This requires to sort and redistribute the dataset in order to maintain full spatial information of a contiguous data chunk. Furthermore, a small overlap (*halo*) of a single layer of ε-cells is necessary to avoid communication with processing cores, clustering the immediately adjacent chunks of the spatial decomposition. Figure 3.3 shows an example of the explained indexing and score-based decomposition step.



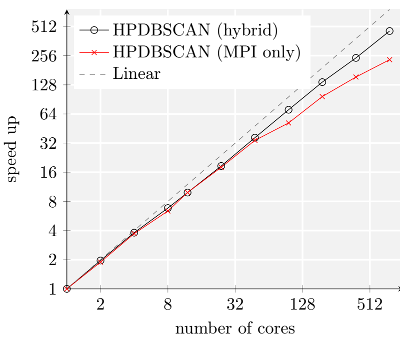
(a) Naïve strategy. (b) Proposed strategy.

Figure 3.3: Spatial domain decomposition strategies for parallelizing DBSCAN. The partition boundary between the processors is shown as a dashed line and the overlapping halo cells in a hatched pattern.

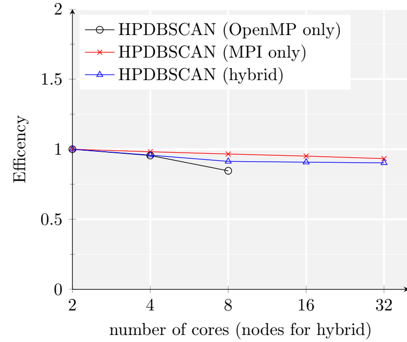
After locally computing the DBSCAN clustering, the partial results need to be merged. This is achieved by finding differences in the cluster labeling in the halos of the domain decomposition. Through transitive cluster ID remapping rules and

a local recoloring step on each processor, the distributed DBSCAN computation results in an equal result compared to a single-threaded execution.

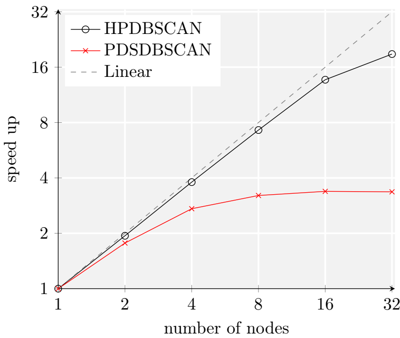
The proposed parallel algorithm is suitable for the execution on both shared- and distributed memory systems and has been implemented as a OpenMP and MPI hybrid in C++. The source code is publicly available and is referenced in the List of Publications. A thorough evaluation of strong and weak scaling properties as well the memory consumption properties of HPDBSCAN has been carried out. The parallelized algorithm outperforms the then state-of-the-art alternative PDS-DBSCAN [52] in terms of computation time, strong and weak scaling and most importantly memory consumption. Figure 3.4 depicts the obtained experimental results.



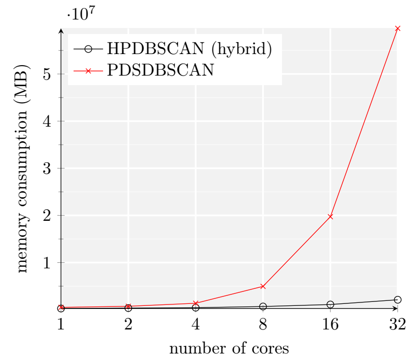
(a) Raw speed-up.



(b) Efficiency [53].



(c) Speed-up comparison.



(d) Memory consumption.

Figure 3.4: Experimental evaluation of HPDBSCAN's performance.

3.3 Paper III: Automatic Object Detection Using DBSCAN for Counting Intoxicated Flies in the FLORIDA Assay.

C. Bodenstein, **M Götz**, A. Jansen, H. Scholz, M. Riedel, "Automatic Object Detection Using DBSCAN for Counting Intoxicated Flies in the FLORIDA Assay", in *Proceedings of 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Los Angeles, USA, pp. 746-751, 2016.

This publication contributes to the third research question in that it demonstrates the applicability of HPDBSCAN for problems outside the Earth Sciences. Furthermore, the reoccurring usage of grid search hyper parameter optimization demonstrates the need for an abstracted implementation in a data analysis library, directly contributing to Research Question Four.

The FLORIDA assay is an experiment currently researched on by the Biological Department at the University of Cologne. In order to be able to understand the involved data analysis problem, the assay is introduced first, before the proposed analysis pipeline is presented later in Section 3.3.2.

3.3.1 The FLORIDA Assay

The FLORIDA assay is concerned with identifying the impact of genetic code and brain structures responsible for alcohol tolerance and, by extension, abuse [54]. Due to similarities in behavior and genetic code compared to humans, the common vinegar fly is used as experimental animal [55]. Genetically altered individuals are exposed to vaporized alcohol before their loss of righting reflex, an indicator for intoxication, is tested. For this, the flies' experimentation container is shaken and observed over time. Vinegar flies with an intoxication level below a certain threshold will automatically fly up towards the lid. Individuals too heavily affected by the alcohol instead are going to gather at the bottom of the container. Based on these two behaviors, a correlation between gene alterations and alcohol tolerance can be derived.

3.3.2 Depth-Separated Image Segmentation

Until recently, these experiments have been conducted manually by laboratory assistants counting the sober and intoxicated flies. This is a error prone and expensive process, which this publication tries to overcome. In this work, an automatic hard- and software pipeline testing the righting reflex in addition to segmenting and counting the intoxicated individuals is proposed. The experimentation apparatus consists of a suspended, perforated plate for the vials, an SLR camera taking images, an electric engine instigating the righting reflex, a backlight and finally a computer connecting the individual parts. Figure 3.5 depicts a schematic overview of the proposed experimentation apparatus.

A software specifically designed for the experiment performs the actual analysis. It can be summarized as an image recognition and segmentation problem with scale

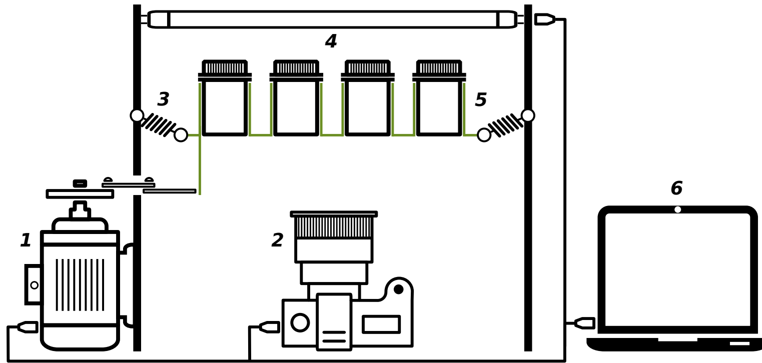
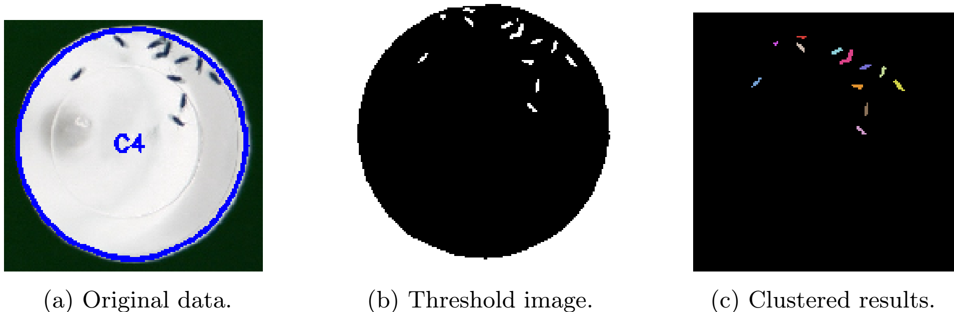


Figure 3.5: Schematic of the FLORIDA assay experiment's hardware setup.

variance—i.e. distance to the focus and image depth separation matters. Traditional techniques like *Scale-Invariant Feature Transform* (SIFT) [56] are not applicable, as they would not be able to distinguish flies at the top from the ones at the bottom of the container due to the scale invariant properties of the algorithm. Fully supervised learning approaches, based on neural networks for example [57], are possible but require a labor-intensive and tedious label creation process.



(a) Original data. (b) Threshold image. (c) Clustered results.

Figure 3.6: Processing stages of the FLORIDA assay depth-perception problem.

Therefore, Paper III proposes a semi-supervised approach, estimating fly count within the experimentation vials using clustering approaches. For this, the vials are first green-screened, then a binary threshold image is calculated and finally the pixels clustered using HPDBSCAN (contribution to **RQ 3**). Only flies that are close to the camera will appear as a distinct cluster due to their size (and therefore pixel density) and result in a counted fly. In comparison to existing image segmentation solutions based on clustering [58], this has the major advantage of being robust to noise, not having to know the cluster count—i.e. the analysis task.

The parameters for the analysis model can in principle be set in the software's interface along other experimentation parameters, such as the vial shaking time and so forth. However, they have been optimized through a MPI-parallelized hyper parameter search (contribution to **RQ 4**), resulting in the experiments default set-

tings. Thereby, the analysis has achieved a mean-squared-error of 1.745, i.e. it is on average off by 1.745 flies in comparison to reality. Figure 3.6 depicts the described analysis stages, which are also visualized for the analysts in the software's graphical user interface. The collaboration partner, the Biological Department at the University of Cologne, has deployed the hardware and analysis script for production in the FLORIDA assay and is using it on a daily basis for months now.

3.4 Paper IV: Automatic Water Mixing Event Identification in the Koljö Fjord Observatory Data

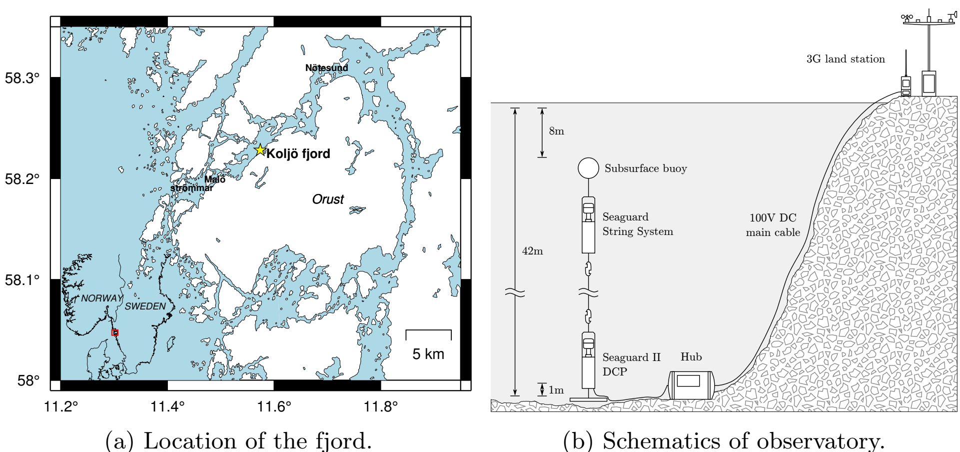
M. Götz, M. Kononets, C. Bodenstein, M. Riedel, M. Book, O. P. Pálsson, "Automatic Water Mixing Event Identification in the Koljö Fjord Observatory Data", submitted to *International Journal of Data Science and Analytics.*

This publication contributes to the second research questions as it shows the viability for an abstracted, parallel grid search optimizer for Earth Science models. In addition to that, it shows that standard analysis algorithm, in this case DBSCAN, should be bundled in a data analysis library.

The Koljö fjord observatory is an oceanographic experiment hosted by the University of Gothenburg, Sweden. Section 3.4.1 will introduce its structure and observation task, before the proposed analysis solution envisioned in this paper is presented in Section 3.4.2.

3.4.1 Koljö Fjord Observatory

The Koljö fjord observatory [59] is located in the eponymous waterway in Sweden. It has a threefold objective. First, testing prototypical measurement technology, second, monitoring of the fjord's health, and third, the study of oceanographic cycles of water exchange between the ocean and rivers. Using anchored underwater sensor equipment the observatory measures various properties of the contained water, such as temperature, salinity, oxygenation and others, which allow the subsequent analysis. One particular problem of interest are so-called water mixing events, i.e. periods of time when water from the ocean or the connected rivers flow into the fjord and change its water properties.



(a) Location of the fjord. (b) Schematics of observatory.

Figure 3.7: The Koljö fjord observatory.

Every half hour the sensors are being read out, resulting in a multi-variate time series data set that has started collection in 2011. Figure 3.7 depicts the locations and a schematic overview of the Koljö fjord observatory including the sensors attached to the string system. This analysis is the first to consider open water mixing event detection, a much more difficult analysis task due to the dynamic system and amounts of noise, compared to that of the previously investigated closed water systems [60, 61].

3.4.2 Water Mixing Event Detection

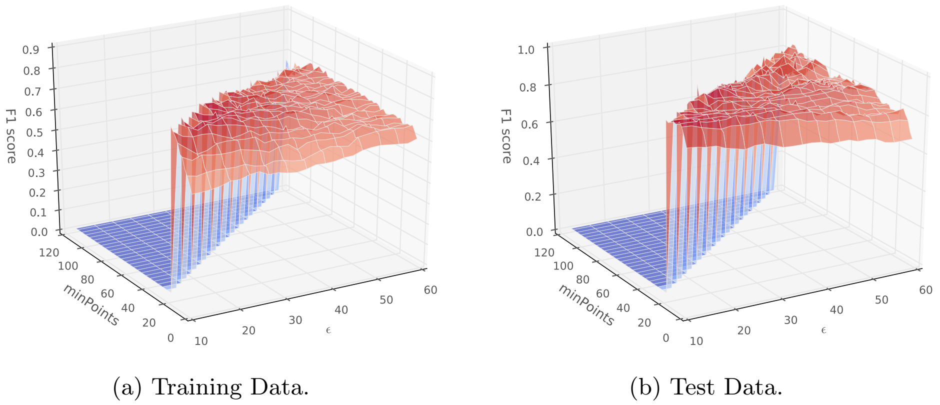
The analysis goal is basically an outlier detection problem in multi-variate time series data. A supervised learning approach is not feasible, because there are no exact per-sample labels. There is still no consensus among domain scientists about how to define a water mixing event start and end. Instead, only the fuzzy center points of the events are known and can be used for validation. This has resulted in the utilization of an unsupervised learning models based on clustering.

After extensive preprocessing, in which the data gaps have been interpolated and smoothed using median and moving-average filters [62], the model suggested in Equation (3.1) has been applied, with Δ^d being the d-step discrete gradient [63], σ the standard deviations, c a confidence interval factor and X the signals. It detects strong peaks or drops in the signals via the discrete time gradient, marks them as univariate outliers if outside a certain confidence interval and subsequently clusters them using HPDBSCAN (contribution to **RQ 2**). Given that a cluster spans a water mixing event's center point, the event is considered detected.

Due to the high imbalance of false-negatives (regular days) compared to true-positives (actual mixing events) the model's performance needs to be measured accordingly. For this, the F1-measure [64] has been selected, as can be seen in Equation (3.2), the harmonic mean of *precision* and *recall* [65]. The model's hyper parameters needed to be optimized, which has been done in a data-parallel fashion using MPI (contribution to **RQ 4**) via the grid-search method. The results can be seen in Figure 3.8, where the model's hyper parameters are related to the resulting performance measure, exposing a crest of optimal values.

$$Events = DBSCAN(\{t | \forall x_t \in X : |\Delta^d x_t| > c * \sigma(X)\}, \varepsilon, minPoints) \tag{3.1}$$

$$F1 = 2 * \frac{precision * recall}{precision + recall} \tag{3.2}$$



(a) Training Data.

(b) Test Data.

Figure 3.8: F1 measure surface for the experimental parameters ε and $minPoints$.

Using the proposed analysis method, an F1-measure of 0.885—with a precision of 0.931 and a recall of 0.843—on unseen test data could be achieved. This analysis is so effective, that the University of Gothenburg, the observatory operator, has begun to deploy the analysis in larger productive use.

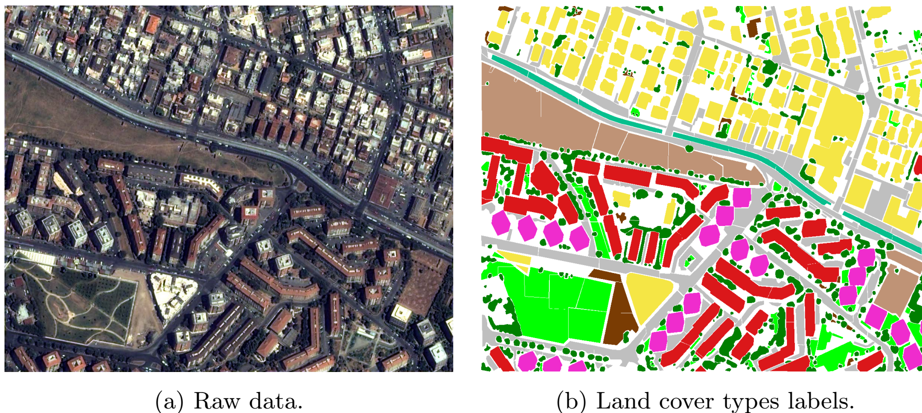
3.5 Paper V: Parallel Computation of Component Trees on Distributed Memory Machines

M. Götz, G. Cavallaro, T. Géraud, M. Book, M. Riedel, "Parallel Computation of Component Trees on Distributed Memory Machines", submitted to *Transactions on Parallel and Distributed Systems.*

This publication contributes to the second research question as it shows how a distributed-memory parallelized max-tree algorithm improves the large-scale analysis of land-cover type classification tasks. Moreover, it contributes to the fourth research questions by demonstrating the recurring patterns of data decomposition and application of sorting routines previously presented in Section 3.2.

The parallel computation of component trees in distributed memory environments is a necessary step in order to enable large-scale feature-engineering for land cover type classification problems. The next two sections are going to introduce the classification problem and, based on that, how features can be derived using component trees, before Section 3.5.3 introduces the contributed parallelization strategy.

3.5.1 Land Cover Type Classification



(a) Raw data.

(b) Land cover types labels.

Figure 3.9: Aerial image of the city of Rome and its land cover types.

Land cover type classification is the problem of assigning a land cover, e.g. road, street, river, etc., to every pixel of a remotely sensed images. These image are recorded by so-called scanning systems mounted to planes or satellites that record the electro-magnetic reflections off of Earths' surface in a push-broom fashion, perpendicular to the flight direction. An example of an aerial image and the corresponding land cover types is depicted in Figure 3.9.

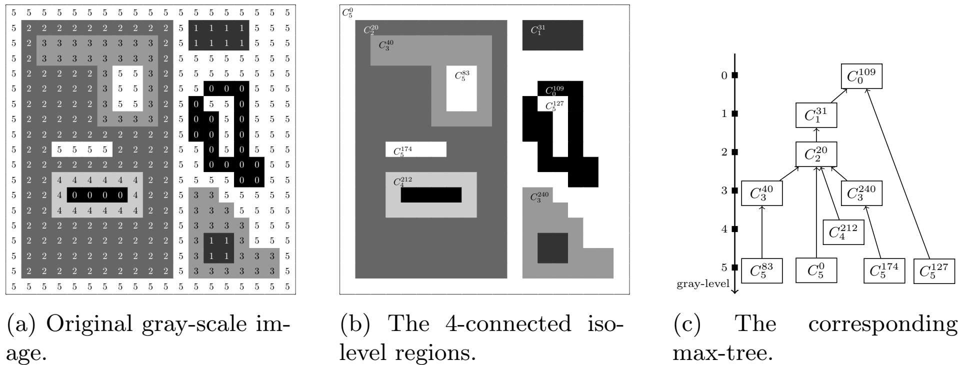
The automated land cover type classification is, for example, used to generate maps, to perform urban planning, catastrophe monitoring and management as well as object identifications in search and rescue and the military. Nowadays, land cover type classification problems are usually treated as supervised learning problem. Probabilistic models based on machine learning, such as Markov random fields [66], SVMs [67] and neural networks [68, 69] form the state of the art with highest prediction accuracy. For the land cover type classification task depicted in Figure 3.9, a one-versus-one [23] multi-class soft-margin kernel SVM [26] of the form shown in Equation (3.3) has been used—with N being the number of samples, i a particular sample or instance, $\hat{y}$ the prediction, x the input data, w a weight matrix to be learned and b the intercept bias and k a non-linear kernel function, e.g. *Radial Basis Function* (RBF).

$$\hat{y} = \sum_{i}^{N} w_i * x_i * k(x, x_i) + b \tag{3.3}$$

3.5.2 Feature Engineering and Component Trees

It has been shown that such a classification task can highly profit from feature engineering [70]. This is a process by which the raw data—i.e. the spectral image pixels—is enriched with artificially created additional bands that include for example spatial context of neighboring pixels. One way to do so is utilizing the mathematical morphology framework and attribute filters. This is a subbranch of mathematics that is concerned with image processing and the extraction of structural and topological information. Component trees is a particular set of algorithm families for dedicated use on gray-scale imagery. Their core idea is to represent image flat zones—i.e. adjacent, connected pixels with the same color value—in a tree, where each level corresponds to a particular gray level of the color depth. The tree has strict mathematical properties that can be efficiently exploited through filter operators, transfering the tree, and with that the image, into a more desirable representation. For example, tree nodes that do not exceed a certain cumulative area can be considered noise and merged with the parent node in the tree.

Figure 3.10 depicts an exemplary gray-scale image for which a particular component tree, called max-tree [71], has been computed. It is a threshold decomposition of an image, starting at the brightest areas up to the darkest areas. This means that for each gray level, starting at the highest, the 4-connected components—i.e. the top, bottom, left and right neighboring pixels—are computed, allowing the connected component operator to traverse higher-colored pixels. The resulting flat zones are then connected in the same order into a tree structure, where each zone points to the next higher-ranking. A formal definition of the max-tree is given in Equation (3.4) [72], where $\mathcal{CC}$ is equivalent to the connected component operator. A min-tree is the inverse of a max-tree, where the tree representation is constructed from the darkest to brightest areas. For practical computation of a min- or max-tree, an algorithm has to effectively solve two graph theoretical problems: connected component labeling [73] as well as graph canonization [74].



(a) Original gray-scale image.

(b) The 4-connected iso-level regions.

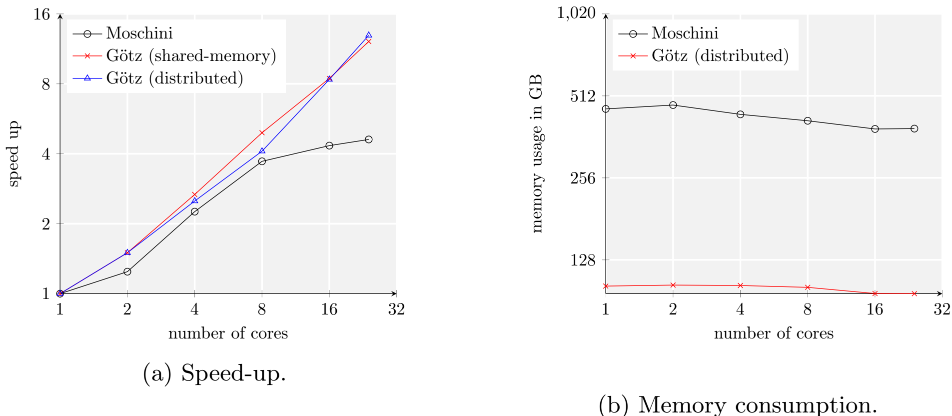
(c) The corresponding max-tree.

Figure 3.10: Example of max-tree representation based on an exemplary image and its components C_i^c, with the subscript c being the gray-level and the superscript i the canonical point uniquely identifying the component.

$$\text{Max-Tree} = \{\mathcal{CC}([Image \geq \lambda]), \lambda \in Pixels\} \tag{3.4}$$

3.5.3 Parallelization Strategy

In Paper V a parallelization strategy for the computation of min- and max-trees in distributed memory environments is presented. Its core idea follows yet again a divide-and-conquer-approach, very similar to the paralleliziation strategy applied in HPDBSCAN presented in Section 3.2. The image is partitioned into equal-sized chunks with a one pixel wide halo region. Each of the partitions is assigned to a distinct processing core, which then computes the local max-tree of the partition. The impartial result is subsequently merged through the resolution of differences in max-tree of the bordering halo region.



(a) Speed-up.

(b) Memory consumption.

Figure 3.11: Experimental evaluation of the distributed max-tree algorithm.

For this, a novel tuple-based merging scheme is employed, expressing each of the tree-connections as a directed as well as inverted link between the flat regions. Iteratively these connections are sorted and merged, i.e. the connected component established, similar as proposed by Flick et al. [75]. Then it is searched for the correct parent component or in other words the graph is canonized. Using this approach the state-of-the-art algorithm has been outperformed in terms of computation time and especially in memory consumption. It also allows for further scaling due to the exploitation of distributed-memory systems. This is useful for the analysis of high-resolution images in land cover type classification, but also for other domains that disallow down-sampling, such as astronomy. Figure 3.11 depicts the achieved results.

3.6 Paper VI: Support Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML Framework

M Götz, M. Book, C. Bodenstein, M. Riedel, "Supporting Software Engineering Practices in the Development of Data-Intensive HPC Applications with the JuML Framework", in *ACM Proceedings of the International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, USA, pp. 1-8, 2017.

This paper contributes to the fourth research question as it presents the requirements and structural components needed for a data analysis framework targeting HPC systems. Its prototypical implementation is already applied in the field and shows promising scalability results.

In Paper VI the *Juelich Machine Learning Library* is introduced—a data analysis and machine learning library and programming framework designed for heterogeneous HPC systems (contribution to **RQ 4**). It is the result of the requirements extracted from the practically analyzed use cases as well as an extensive literature and framework study (see also Section 4.1). JuML's main features include readily implemented, scalable standard machine learning algorithms, such as k-means or artificial neural networks, a distributed data load, and storage access abstraction that employs lazy loading strategies. The latter is designed with parallel file systems in mind and therefore utilizes the *Hierarchical Data Format 5* (HDF5) [76], which is widely used in HPC, as I/O format of choice. Built-in routines allow to chunk computational workloads, here the data items, and also enables the access to halos—all selected by the processing algorithm.

Moreover, JuML has the capability to execute compute kernels on traditional CPUs as well as accelerators such as GPGPUs or OpenCL-capable FPGAs. This is achieved by employing ArrayFire as numerical and tensor computation library. ArrayFire has vectorized, single-threaded, OpenMP-, OpenCL- and CUDA-parallelized implementations of basic mathematical functions, such as matrix summations, multiplications, etc. Using this approach the development of more complex analysis for heterogeneous computation backends is significantly simplified. However, there is still a need for detailed attention by the developers to achieve highly effective code on all of the available execution environments. Therefore, complex, however recurring, low-level routines, such as distributed sorting or class label normalization for datasets, are pre-implemented as part of JuML's framework. This does not only minimize development time for new data analysis algorithms, but also eases the realization of scalable solutions.

JuML's core is written in C++ in order to enable low-level code optimizations, especially with respect to hardware aspects. Via the interface generator SWIG [77] Python bindings are automatically generated. Being the language of choice in data analysis, it exposes a familiar programming environment and a simplified API that

is more suitable for rapid prototyping and a data exploration workflow. Using SWIG, one could in principle also create interfaces to other languages, such as for R [78] or Julia [79]. This, however, was not in the scope of this publication.

Instead, it is focused on another main contribution, which is the discussion of software engineering aspects in the design of an HPC data analysis framework. Therefore, a strong emphasis is put on unit testing in the context of parallel code, which is executed on different hardware backends and in distributed environments. The challenge is to keep all computations numerically stable and deterministic, independent of the processing platform's configuration. Therefore, the publication introduces a self-developed testing toolkit used in line with JuML. It enables the generation and execution of multiple unit test permutations, taking into account accelerators, shared- and distributed-memory parallelization. For this the build system CMake [80] and the testing framework Google Test [81] are employed and customized for the application use case.

The paper concludes with a performance evaluation of analysis algorithms developed with JuML. In this case, the land cover type classification use case, introduced in Section 3.5.1, has been revisited. Using JuML's neural network implementation, executed on multiple GPUs, the analysis achieved substantial speed-up while maintaining the prediction accuracy and code-length. Due to its success, JuML has been selected as benchmark framework for the data analysis module of the experimental exascale HPC prototype system DEEP-EST. At the same time, it is also the conceptual template for the Helmholtz Analytics Framework, a Germany-wide initiative for the design and implementation of a large-scale data analysis framework, satisfying the technical needs of the partaking research institutions and their domain use cases.

The following Chapter 4 scrutinizes JuML and its core design concepts more detailed. In particular, the data distribution and parallelization strategies are highlighted and put into context with a set of proposed general requirements for an HPC data analysis framework.

Chapter 4

An HPC Data Analysis Framework

This chapter lays out the author's vision of a data analysis framework and algorithm library, targeting heterogeneous, distributed high-performance computing systems. The presented design proposition is rooted in the author's own experience based on the case studies as well as an extensive literature study. Section 4.1 reviews the existing collections of computational solutions commonly in use across the data analysis domain and the properties relevant to their intended application in HPC as well as acceptance in the user communities. Commonalities recognized in the literature, existing software products and own engineering undertakings in applied studies presented in the publications, resulted in a generalized perspective on the ways to approach two major types of challenges: (1) data analysis algorithm parallelization and (2) requirements for a framework enabling applied Big Data analysis studies.

Despite algorithmic dissimilarities between distinct data processing and analysis methods, the same uniform internal parallelization strategy applies, given that scalability across the data amount is the prime objective. This parallelization paradigm is rationalized in Section 4.2, which outlines the aspects one must consider when parallelizing such algorithms, as well as ways of addressing them. Following this paradigm supports further algorithm development not only by providing structure to the process, but also by identifying recurring basic functions and building elements, taking away major amounts of otherwise necessary engineering workload.

The principles of reusability and abstraction, which lie at the foundation of engineering progress, also need to be applied in the data analysis case studies. On an abstract level, the sequence of intermediate steps is unchanging as given by the data analysis process highlighted in the background Section 2.1.3. Analogously, the applied methods and concrete algorithms also stem from a set of standard approaches, such as DBSCAN or SVMs, as exemplified in the publications of the use case studies and related work in the field of applied data analysis.

Therefore, it is argued that the creation of a framework for, and a library of pre-implemented parallel and scalable solutions, ready to be used for Big Data analyses

on HPC systems, is the next logical step. In practice, this framework should allow analysts to focus on the tasks relevant to their primary field of expertise, rather than the technical aspects. Therefore, a list of feature requirements for a general-purpose data analysis framework for heterogeneous HPC systems is presented in Section 4.3. It points out the essential components, necessary for supporting an analyst in each step of the data analysis process from start to end.

Finally, the chapter is concluded in Section 4.4 by presenting the *Juelich Machine Learning Library*, a prototypical design and implementation of said requirements list, translating it into a tangible and usable HPC data analysis framework.

4.1 Related Work

It is widely known that there are already a number of data analysis and machine learning libraries available. Majority of them, however, are not designed for dealing with large data amounts, let alone parallelization for accelerators or distributed-memory systems. Nevertheless, due to their long existence these libraries are popular and in wide-spread use. A non-exhaustive list can be summarized as follows.

For C++, *SHOGUN* [82] offers powerful tools for both application scenarios: general-purpose data analysis and machine learning. Through SHOGUNS's interface for Python it also supports a more explorative or rapid-prototyping workflow. The benefits of this interface deserve a particular emphasis, as Python is favored in the HPC community and specifically also in the field of data analysis. As a result of broad support and stable interest, this programming language features a large number of third-party analysis libraries. Among them are, for example, *scikit-learn* [83], *scipy*, *numpy* [84], as well as *pandas* [85].

For Java, the popular WEKA [86] machine learning suite offers a wide variety of readily available algorithms. Moreover, a graphical user interface enables the visualization of typical statistics for loaded data sets and can also generate plots. On the other hand, WEKA suffers from a number of performance limitations in terms of computation speed and memory consumption.

Finally, R, Matlab and Octave, while being in their essence programming languages, have comprehensive standard libraries including routines designed for statistics, data analysis and machine learning. Especially in the fields of signal processing and engineering, these languages are predominant. Similarly to the libraries named above, the initially designed packages were not made with parallel execution environments in mind. However, parallel processing capabilities are being added at the cost of substantial engineering efforts. R, for example, has been recently extended with MPI bindings, while Matlab has undergone a major update late in 2017, adding GPGPU-accelerated neural network implementations.

Generally, neural networks have seen a large influx of frameworks and libraries in the last years due to the broadening research and industry interest in deep learning. Due to the matrix-matrix-multiplication-heavy nature of neural networks, concrete implementations utilize shared-memory-parallelized vector-operations and accelerators like GPGPUs to speed-up the computation. In the machine learning area, the three major matrix- and tensor-numerical frameworks, implementing highly efficiently parallelized operations, are: Google's TensorFlow [46], Theano [87] and

Torch [88]. While these tensor frameworks are not strictly limited to neural networks, up until now this is the prevalent application domain.

On top of this, there are dedicated neural network and deep learning programming libraries, either utilizing the above numerical frameworks of low-level, mathematical operations, or even implementing their own. Keras [89], for example, is a Python-based project that wraps both, TensorFlow and Theano, as compute backends, and provides an easy-to-use, high-level API and convenience functions. A combination of both, a matrix compute engine and high-level neural network API, can be found in the following frameworks: MXNet [90], Caffe 1 and 2 [91], (py)Torch [92] and CNTK [93]. A number of these are supported or directly contributed to by major players in the area of machine learning and artificial intelligence, such as Baidu, Facebook, Microsoft and Yahoo, and have been released mostly within the last two to three years. These frameworks are highly advanced and even support multi-GPGPUs, and distributed model training and classification. One can consider this area saturated with well-established and well-engineered solutions.

Still, not all data analysis tasks can be tackled efficiently with neural-network-based machine learning. They require different solutions, the major ones here being data preprocessing, unsupervised clustering models, tree and ensemble methods, feature engineering and data augmentation, as well as hyperparameter optimization and model validation. Under the consideration of these aspects, including the provision of pre-implemented parallel and scalable algorithm implementations, the availability of solutions thins drastically. MLPack [94] is a C++ framework that offers a variety of these features, but is only selectively shared-memory-parallelized. Intel's *Data Analytics Acceleration Library* (DAAL) is similar in that regard, but offers even more features and additionally Python, R and Matlab bindings. Moreover, it is able to exploit distributed computational resources when used in conjunction with map-reduce. Therefore, it can be considered to be a software solution targeting HTC systems not directly usable on HPC systems. The utilization of MPI is in principle possible as well, but must be custom-developed by the respective DAAL user.

A notable mention that is purely designed for HTC systems is the MLlib [47]. It ships with the Apache Spark map-reduce stack. Similar to the previous frameworks, it contains data analysis models and features outside neural network-based learning, yet it is highly limited in the complexity of the algorithms due to the framework's communication design. This is, for instance, reflected in the fact that it only supports linear kernel SVMs, or binned decision trees.

As a closing remark, a topic that has not been widely addressed in terms of available software is hyperparameter optimization. It is expected to become increasingly more relevant in the next years due to the sheer amount of different model approaches and their flavors [95]. Grid search, the most basic and not very resource-efficient method, can be easily implemented in an independent, parallel fashion. However, with the growing number of parameters, it will be necessary to find effective optimization schemes, based on evolutionary algorithms, particle swarm optimization, etc. Hyperopt [96] is the prime example for such a distributed solution based on the NoSQL database MongoDB.

4.2 Data Analysis Algorithm Parallelization Paradigm

The similarities in the parallelization strategies employed in HPDBSCAN (Section 3.2) and in the distributed max-trees (Section 3.5) indicate that there is a more general paradigm in formulating highly scalable parallel data analysis algorithms. Generally, both presented approaches follow a modified version of the well-known concept of divide-and-conquer algorithms, in that they:

1. Split the problem, here the data set, into two or more sub-problems,
2. solve the subdivided problem using the standard non-parallelized data analysis kernel, and then
3. recursively merge the partial results, utilizing halos when necessary.

This statement makes two assumptions about the nature of the scalability of the analysis task at hand. First, the analysis kernel is the time- and computation-intensive bottleneck, for which the utilization of parallelization is beneficial to begin with. Second, the data set can be easily decomposed into sensible sub-problems that can be independently analyzed, and for which a meaningful partial result can be obtained.

In data analysis and machine learning, the latter assumption, with a few exceptions, is often true. The analyzed data sets in question are usually matrices, time series, images, volumes, batches of these and so forth. Therefore, a decomposition can be naturally performed along the data's major dimension, e.g. the matrix rows, the time axis, or the individual instances in a batch. Consequently, it is possible to generalize the data partitioning and distribution step in cases where the data is to be analyzed in parallel using multiple processors or even nodes. Figure 4.1 illustrates this relationship schematically with an exemplary three-dimensional data set, decomposed along the major axis, also called samples.

Putting it into the perspective of a general-purpose data analysis framework for heterogeneous HPC systems, would make it possible to abstractly define, implement and on-demand utilize a number of predefined data access distribution strategies. The list of applicable strategies would be as follows: (1) a balanced strategy, where each processor gets the same amount of data items for homogeneous systems, (2) a weighted variant of this for heterogeneous compute platforms, where the processors have different relative computational power, as well as (3) a flavor for each of these two, additionally providing a halo, easing the ensuing implementation of merging strategies.

Depending on the data, it might be necessary, after loading the partial data, to sort, redistribute and balance the data chunk assigned to each of the processors, in order to avoid further communication and to balance the workload. This is typically the case for data that is either sparse and highly skewed, or where one processor does not have complete information within the assigned data chunk. HPDBSCAN resorts to this approach for spatially dispersed point clouds.

Given that each processor now has a coherent data chunk, it is possible to compute a partial analysis result of the decomposed data using the standard implementation of the analysis algorithm. As a result, there is no necessity to invest

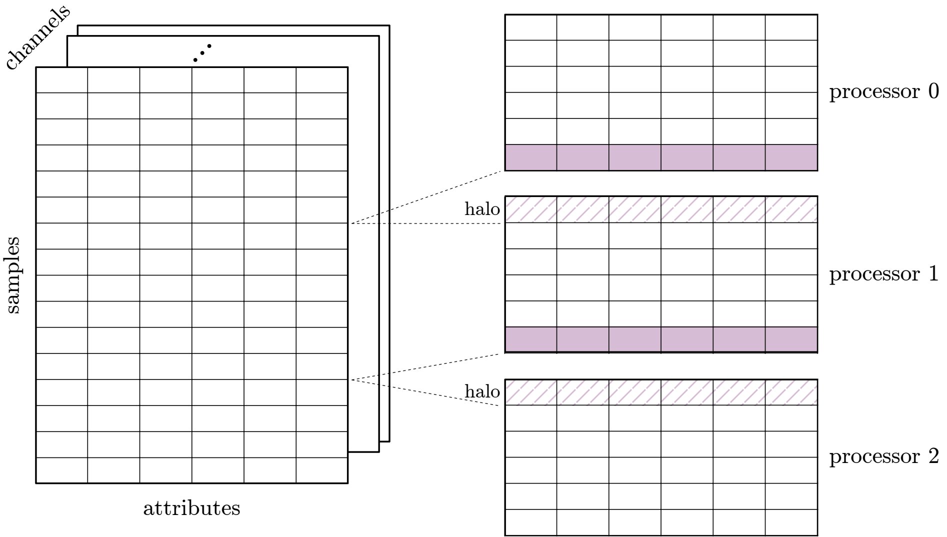


Figure 4.1: One-dimensional data decomposition across the samples, including halo zones in purple and duplicates with a hatched pattern.

engineering efforts into modifying the algorithm itself. The challenging part is to merge the obtained partial results into a consistent global view across all nodes (compare to the divide-and-conquer step three). This cannot be done in a generalized fashion and is inherently dependent on the analysis algorithm that is to be parallelized. However, there are commonly reappearing sub-problems, such as sorting, recoloring and so forth, which could be provided by a data analysis framework to assist the implementation of a scalable algorithm. Merging the partial results might involve a significant data exchange between the processing cores with highly complex communication patterns. Therefore, the proposed parallelization paradigm is particularly suitable for HPC systems with high-bandwidth interconnects and flexible message passing capabilities, contrary to HTC approaches [97].

Algorithm 1 depicts the pseudo-code of a proposed general scalable data analysis parallelization paradigm based on the made observations. Following this paradigm makes it arguably easier to formulate scalable data analysis algorithms, especially when dedicated for distributed-memory environments. This is supported by the fact that other well-scaling data analysis algorithms for distributed-memory environments, even though not explicitly referring to this paradigm, function this way. Prominent examples include CascadeSVMs [98], distributed watershed filters [99] or distributed multi-GPGPU gradient-learning for neural networks [100]. Therefore, the presented data distribution strategies, as well as the mentioned low-level support routines for merging partial results, are essential components for a scalable data analysis framework for heterogeneous HPC systems and can therefore be found on the requirements list in Section 4.3.

Algorithm 1 Pseudo-code of the data analysis parallelization paradigm.

```
1: @parallel
2: procedure parallel − analysis(data_set)
3:     n ← number of nodes
4:     r ← processor id in range [0, n[
5:     t ← number of threads
6:
7:     if Is-Master(t) then
8:         partial − data ← Load-Partial-Data(data_set, r, n)
9:         halo_sizes ← Decompose-Domain(partial_data, r, n)
10:        halos ← Load-Halo(data_set, halo_sizes, r, n)
11:    end if
12:
13:    if Incomplete(partial_data) then
14:        partial_data, ordering ← Sort-And-Balance(partial_data)
15:    end if
16:    partial_solution ← Local-Analysis-Kernel(partial_data, halos, t)
17:
18:    partial_rules ← Resolve-Halos(halos, partial_solution, r, n)
19:    global_rules ← Reduce(partial_solution, partial_rules, r, n)
20:    global_solution ← Apply(global_rules, partial_data)
21:
22:    if Is-Master(t) then
23:        global_solution ← Redistribute(global_solution, order)
24:        Store-Results(global_solution)
25:    end if
26: end procedure
```

4.3 Requirements

Based on the data analysis process introduced in Section 2.1, the properties of modern HPC systems described in Section 2.2, and the generalized algorithmic paradigm laid out in Section 4.2, one can derive the requirements for a scalable data analysis framework for HPC systems as follows.

- The framework should provide an abstracted interface for handling distributed data sets, which is able to load common HPC data formats, like HDF5 or netCDF, automatically exploiting parallel I/O. At the same time this entity must be capable of supporting various common data distribution strategies based on the applied data analysis algorithm.

- A library of standard data analysis and machine learning algorithms used for studies in use cases is necessary. Among them should be pre-processing algorithms such as statistics or image filters, unsupervised learning models, like clustering algorithms, as well as supervised learners, e.g. SVMs, (deep) neural networks or decision trees.

- There should be the possibility to perform automatized hyperparameter optimization of the models, based on grid search in the simplest case, or advanced strategies, such as evolutionary models, for more complex analysis models. This naturally includes the provision of appropriate model performance evaluation measures such as accuracy, F1 or ROC.

- The inclusion of high-level, non-verbose language interfaces, such as Python or R, that support the rapid prototyping workflow in the initial explorative stages of the data analysis process is desired.

- Such a framework needs to support different common compute backends present in modern HPC systems. This includes, at the bare minimum, CPUs and GPGPUs, but additional support for FPGAs and MICs would be beneficial.

- For the implementation of further data analysis algorithms, commonly reappearing components should be pre-implemented in a highly efficient manner. Examples include class label normalizers, efficient distributed sorting routines, histogram calculations and data workload balancing.

- Finally, recurring convenience functions outside the main analysis model construction step, such as train- and test-data division, computation of statistics, or the generation of sliding windows across the data set, would ease and encourage the development of analysis programs using the framework.

It should be clarified that in practical analysis applications the data analysts mainly focus on the four steps: exploration, preprocessing, analysis and evaluation of the data analysis process. The data collection and product deployment phase is usually assumed by the collaborating domain scientists. Therefore, these aspects have not been considered in the definition of the requirements, but might be necessary in distinct real-world scenarios, adding to the requirements for custom cases.

4.4 Juelich Machine Learning Library

The *Juelich Machine Learning Library* is a prototypical implementation of a data analysis framework for heterogeneous, distributed HPC systems. While it is not entirely feature-complete, it addresses all of the devised requirements listed in Section 4.3. In line with them, JuML includes implementation for each of the essential components to perform full-fledged use case analyses. Section 3.6 has already briefly summarized the major engineering aspects that have been incorporated in the framework's design, published in Paper VI. However, JuML's general structure and intended workflow, which shall be the subject of this section, has not yet been portrayed.

The central entity a data analyst works with when using JuML is the `Dataset` class (see UML class diagram in Figure 4.2). It enables to load, store and transform data. Every preprocessing step and data analysis algorithm receives a `Dataset` object as input and conversely generates one as output. This enables analysts to actively think in terms of data flows, while at the same time presenting framework developers with a standard data container abstraction. Generally, a `Dataset` can be thought of as an abstract handle to the actual underlying data, which may very well be distributed across multiple processing nodes.

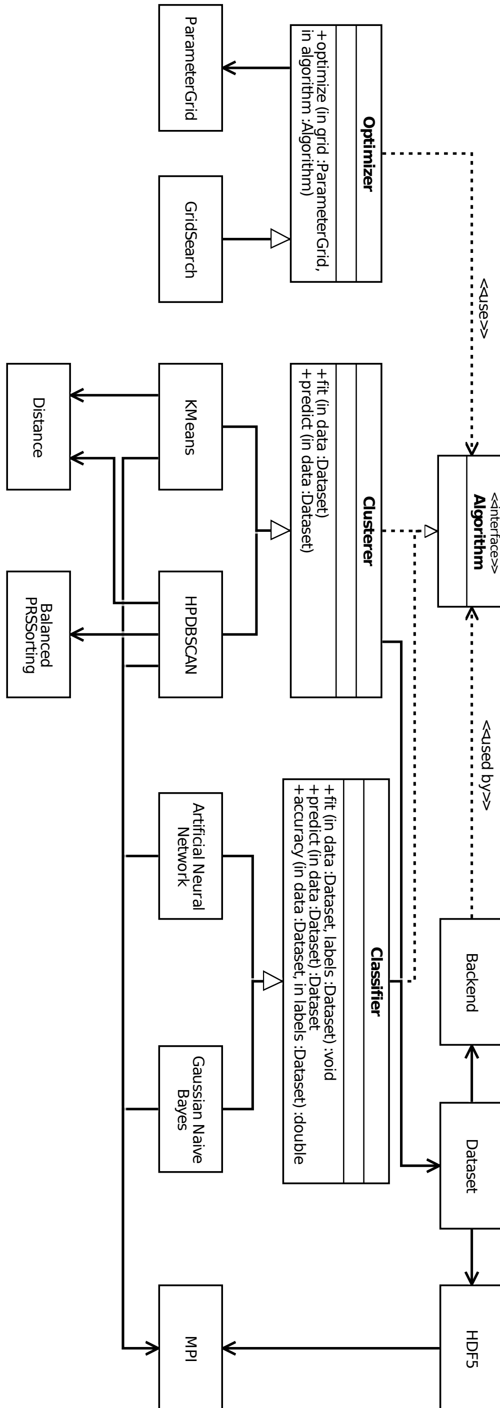


Figure 4.2: UML class diagram [1] of JuML's system structure.

A `Dataset` object can be initialized either from a pointer local to the memory of the processor, or from a data set[1] within an HDF5 file. The choice for HDF5 is motivated by the more wide-spread use of this format in the examined application use cases, compared to other formats like netCDF, as well as its support in other data analysis frameworks, e.g. Keras. Generally, the latter data access strategy using HDF5 is preferred over the raw data ingestion, because it allows to delay the decision how the data needs to be distributed until a particular algorithm is executed on it. Using this lazy loading approach, JuML only requires to store the file and HDF5 data set path, up until an algorithm directly chooses a correct domain decomposition strategy (see also Section 4.2), without having to potentially redistribute the data like in the pointer-case. In the current implementation state JuML applies the lazy loading feature, but only supports the simplest equi-chunked distribution strategy without halos.

Internally, the data is then converted into tensor data structures offered by the ArrayFire numerical library, enabling the computation on heterogeneous compute backends. JuML implements the actual kernels of the data analysis algorithms using ArrayFire's API and applies them to the data portion local to a machine. The distributed data exchange during the merge-step of the partial results (compare to Section 4.2) is realized through MPI. By default, JuML is greedy for the distributed, inter-node parallelization, setting the MPI communicator to all available resources—`MPI_COMM_WORLD`—while it is conservative for the local intra-node parallelization, selecting the host CPUs as computation backend, ensuring the maximum amount of parallelization while guaranteeing the execution. However, the allocation of the workload may be customized by the user. For this, all analysis algorithms offer to pass two additional parameters, besides the one parametrizing the analysis itself, to the object implementing it, which can carry a symbolic handle for the computation backend as well as an MPI communicator. Listing 4.1 shows an example for a k-means clustering.

```
1  import juml
2  from mpi4py import MPI
3
4  data = juml.Dataset('example.h5', 'my_data') # path and data set name
5
6  clusterer = juml.KMeans(
7      k=3, # clusters to be identified
8      backend=juml.Backend.CUDA, # intra-node backend (CUDA)
9      comm=MPI.COMM_SELF # inter-node allocation
10 )
11 clusterer.fit(data)
12
13 print clusterer.centroids()
```

Listing 4.1: Usage example of JuML's Python API—instantiation of a k-means clustering algorithm, executing the kernel on GPGPUs and a single node only.

[1] A data container specified in HDF5; not to be confused with the `Dataset` class.

The full list of features already implemented in JuML is listed below:

- Distributed data sets
  - Lazily loaded HDF5 I/O
  - Raw data pointer
- Support of heterogeneous compute backends
  - CPUs with OpenMP-based parallelization
  - CUDA-capable compute architectures
  - OpenCL-capable devices
- Preprocessing algorithms
  - Class label normalization
  - Dataset normalization (fixed intervals, standard deviation)
- Artificial neural networks
  - Directed, acyclic architectures
  - Fully-connected, dense neurons
  - Linear, sigmoid and hyperbolic tangent activation functions
  - Batched gradient descent back-propagation optimizer
- Gaussian naïve Bayes classifier
- K-means clustering
  - Random and preset centroid initialization
  - Optimized k-means++ initialization [101]
  - k-median [102] variant of centroid determination
- Early portation of HPDBSCAN
- Grid search hyperparameter optimization
- Auxiliary low-level routines
  - Euclidean and Manhattan distance metrics
  - Distributed parallel sorting through regular sampling

As mentioned before, JuML is already in productive use in two major scientific projects. First—the prototypical exascale research cluster DEEP-EST—utilizes the framework to benchmark the data analysis sub-system, and second—the Helmholtz Analytics Framework, a Germany-wide initiative to establish a common HPC data analysis and machine learning library—uses JuML as conceptual template.

Chapter 5

Conclusions

The scientific community's interest in large-scale data analysis has been continuously increasing over the last decades, with innovations in data-driven model construction and in-situ processing of simulation results attracting particularly intensified demand. It is expected that this trend will not cease in the near future, but rather persist. A review of the state of art in data analysis algorithm development in computational science literature displays numerous examples of initial adaptations to the challenges posed by Big Data. Nevertheless, the lack of and pressing need for a systematically developed, scientifically scrutinized and openly available base of analysis tools, on par with the available technology, is evident from the conducted field survey. This, along with in-depth investigation of selected scientific use cases, motivates the author's proposal to develop a large-scale data analysis framework for HPC systems. In course of doctoral studies, a first candidate implementation, in form of a framework prototype called JuML, has been suggested, attempting to bridge the apparent engineering gap.

Research problems tackled in this thesis were expressed in form of four major research questions, providing guidelines for the analysis and discussion of the performed case studies. This dissertation's subject matter lies at the intersection of distinct fields of science. Each of its case studies documents the use of practices from the domain of computational engineering to speed-up the experimentation and data analysis in a specific Earth science domain. While any particular one publication provides insights into the successfully executed efforts at boosting parallelization and scalability in a conventional data analysis algorithm, the findings extracted across the collection of studies lead to more universal conclusions, pertaining to key aspects of the computational data analysis domain development. Supported by the following research questions, the thesis lays out the synergies of Earth sciences and computational engineering in the age of HPC, and ways to advance them.

RQ 1: What is the state of the art in data analysis and its technologies in the Earth sciences?

RQ 2: How and which parallel and scalable algorithms can support the analysis of selected Earth science use cases?

RQ 3: Can the identified techniques be applied in a generalized fashion in scientific domains outside of the Earth sciences?

RQ 4: Are there enough commonalities that would justify the design of a parallel and scalable data analysis framework and algorithm library for HPC computing systems?

Publications included in the thesis answer or contribute to answering these questions. Paper I, a technical survey of parallelized implementations of non-linear SVMs and the clustering algorithm DBSCAN, tackles the first research question. While the investigative focus targets two specific analysis algorithms, with numerous possible alternative methods at hand, at the basis the major training approaches in machine learning—supervised and unsupervised learning—have been tackled. Moreover, both of the explored algorithms are in widespread use and are essentially standard tools for analysis tasks. Given that even such currently sought-after, yet long established, algorithms have no readily available scalable implementations, the state of the art is judged as arguably deficient, pointing to potential for improvement. This position is further reinforced by the systematic data analysis framework and library review given in Paper VI and in Section 4.1 presented in this thesis.

Papers III, IV, V and to some degree also VI investigate particular scientific domain use cases using parallel and scalable analysis tools. Each on its own constitutes a contribution to the respective analysis application fields and respond to the posed **RQ 2** and **3**. The case studies represent a careful selection of distinct scientific challenges, involving a variety of analysis tasks, data set characteristics and spanning a diverse range of approaches. These include:

1. Unsupervised, semi-supervised and supervised analysis tasks using,
2. clustering, classification and outlier detection approaches in
3. simple and hyperspectral images, multi-dimensional point clouds and time series data.

The expertise gained from these studies is used to guide the design of the proposed framework allows for a generalization to other domains, and through that contributes to addressing **RQ 4**. Conversely, three major data analysis aspects have not been explored in this dissertation, while they would be a meaningful addition to the list above. These are, (1) general regression problems, e.g. forecasting measured values, (2) cohort data and data set combinations, like time series of images for example, and (3) reinforcement learning models. Due to time constraints, it was unfeasible to investigate all of these problems. Notwithstanding such objective limitations, in an already ongoing study, the author investigates a ozone concentration forecasting problem using neural networks and reinforcement learning, to overcome this shortcoming of the thesis.

Finally, the last major contribution of this thesis are parallelization strategies for data analysis algorithms and, their concrete implementation, as open-source software. HPDBSCAN and the distributed max-trees algorithm are general-purpose

data analysis algorithms that have been applied in different domains, contributing to **RQ 2** and **3**. Moreover, the commonalities in the parallelization strategies also enabled the formulation of a more abstract parallelization paradigm, which is an input to **RQ 4**. This is also one of the key design features for JuML, the HPC data analysis framework presented in this thesis.

As laid out, the major steps in developing well-scaling data analysis algorithms are essentially the same, following an abstract paradigm, with the merging step of the parallel computation being the only part that has to be customized. Thus, JuML has abstracted and encapsulated recurring components. This ranges from low-level routines, like sorting, to data set distribution as well as complete algorithm implementation. While the parallelization aspects in the JuML's code are not entirely hidden from the data analysts, they are, however, reduced to two simple handles to be used for the inter- and intra-node workload distribution. As a result, applying highly computationally efficient algorithms that exploit the available hardware capabilities to a large extent, in line with its core purpose, becomes transparent.

Due to the fact that JuML is still only a design prototype, it has shortcomings in the number of pre-implemented data analysis algorithms it offers. Nevertheless, due to the decision to adopt it as the benchmark framework for the data analysis module of the prototypical HPC system DEEP-EST, and as the conceptual template for the Helmholtz Analytics Framework, it is expected that the number of features is going to steadily increase in the immediate future.

5.1 Future Work

A number of possible research directions, suggested by the distinct studies presented in this thesis, remain to be explored in the future. In the data analysis use cases shown here, the investigative methods involved clustering used in a (semi-) supervised fashion, with a degree of fuzziness in the data set labels or patterns to be learned from. With the vigorous development trends of neural network techniques, the emerging methods of deep and policy learning could be used to reapproach the presented challenges [103]. The problem of object detection in point cloud might, on the other hand, be suitable for unsupervised learning with SOMs [29]. Even though such data analysis methods often display a certain robustness to imprecise input, fuzziness tends to bias the evaluation of learning performance, making the representativeness of its outcomes suffer. Mitigating such bias is of foremost importance, especially in comparative studies, and presents another challenge to be faced in future work. The endeavor to overcome them is often difficult, albeit necessary, to improve the quality of the data set labels, or providing explicit definitions of analyzed patterns, e.g. using an appropriate fitness function. The former especially applies to the fly segmentation problem [104], while the latter could greatly facilitate the Koljö fjord observatory study [105] by reinforcing the data set labels with descriptive or explicit definitions of water mixing event boundaries.

A potential roadmap for further research on parallelization strategies for DBSCAN is more straightforward. There are already groups working on enhancing the HPDBSCAN's proposed parallelization strategy presented here. The approaches

range from simple code and communication primitive optimization to portations to different hardware platforms [106]. A more general and highly viable approach could be to exchange the regular local DBSCAN clustering step with an enhanced version called AnyDBC [107]. This concept computes deterministically the same results, but gains significant speed-up by building high-level cluster structures of dense hotspots first and then combining these via the density connectivity. As a result, one can expect a significantly shorter local clustering computation time for HPDBSCAN. As of now, however, AnyDBC is not yet shared-memory parallelized, presenting yet another research opportunity. Alternatively, one could try to parallelize OPTICS [108], an iterative DBSCAN sub-space clustering algorithm, which would enable group identification in high-dimensional data sets.

Future work on the distributed max-tree should follow three main objectives. Firstly, in line with the original motivations, it should be incorporated into the extended *Self-Dual Attribute Profile* (SDAP) pre-processing method, designed by Cavallaro et al. [70], and in conjunction with it, used for the analysis of gray-scale images in land-cover type classification, brain region segmentation or astrological star cluster detection. Secondly, in order to make practical use of the generated max-tree, for instance in image manipulation, it would be beneficial to also allow the distributed computation of attribute filters, i.e. operations that modify the tree. Initial works in this direction are presented by Moschini et al. [109], but only consider simple area filters. Thirdly, the algorithm proposed in this thesis could be extended to the *Tree Of Shapes* (TOS) [110]—a contrast-invariant component tree, which computes the min- and max-tree alternatingly. This representation has more powerful analysis capabilities and could therefore increase prediction accuracy in challenging classification tasks [111].

Finally, the future of JuML, the author's most tangible domain contribution, is being actively shaped. It has been chosen as the benchmark suite for the data analysis module of the experimental DEEP-EST cluster system [112], a three-year EU project for testing exa-scale HPC systems. Proceeding with that aim, new data analysis algorithms will be developed, consistently integrated, as well as extensively benchmarked in terms of their weak and strong scaling. Moreover, the Helmholtz Association—the largest union of scientific organizations in Germany—counting 18 members, has newly launched the HAF initiative [113]. JuML has been selected as the prototypical basis for the project. It will most likely undergo some changes, such as exchanging ArrayFire [45] with TensorFlow [46] due to more widespread adaption. However, most of the already existing code can be ported with little effort, due to high-level operation abstraction in both frameworks. One of the major expectations is a permanent installation and usage of JuML's HAF successor in the Helmholtz compute facilities.

References

[1] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[2] Min Chen, Shiwen Mao, and Yunhao Liu. Big Data: A Survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[3] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data Mining with Big Data. *IEEE Transactions on Knowledge and Data Engineering*, 26 (1):97–107, 2014.

[4] Hsinchun Chen, Roger Chiang, and Veda Storey. Business Intelligence and Analytics: From Big Data to Big Impact. *MIS Quarterly*, 36(4), 2012.

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

[6] Michael Diepenbroek, Hannes Grobe, Manfred Reinke, Uwe Schindler, Reiner Schlitzer, Rainer Sieger, and Gerold Wefer. PANGAEA—an Information System for Environmental Sciences. *Computers & Geosciences*, 28(10):1201–1210, 2002.

[7] Peter Dewdney, Peter Hall, Richard Schilizzi, and Joseph Lazio. The Square Kilometre Array. *Proceedings of the IEEE*, 97(8):1482–1496, 2009.

[8] Doug Laney. 3D Data Management: Controlling Data Volume, Velocity and Variety. *META Group Research Note*, 6:70, 2001.

[9] Wullianallur Raghupathi and Viju Raghupathi. Big Data Analytics in Healthcare: Promise and Potential. *Health Information Science and Systems*, 2(1): 3, 2014.

[10] Mark Van Rijmenam. *Think Bigger: Developing a Successful Big Data Strategy for Your Business*. AMACOM a Division of American Managmenet Association, 2014. ISBN 9780814434154.

[11] Kirk Borne. Top 10 Big Data Challenges a Serious Look at 10 Big Data V's. `https://www.mapr.com/blog/top-10-big-data-challenges-look-10-big-data-v`, 2014. [Online, Accessed: 2017-08-31 12:21].

[12] Tom Shafer. The 42 V's of Big Data and Data Science, April 2017. URL `https://www.elderresearch.com/company/blog/42-v-of-big-data`. [Online, Accessed: 2017-08-31 18:09].

[13] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *26th symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[14] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, 2012. ISBN 9781491901632.

[15] James G Shanahan and Laing Dai. Large Scale Distributed Data Science using Apache Spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2323–2324. ACM, 2015.

[16] Robert Flowers and Charles Edeki. Business Process Modeling Notation. *International Journal of Computer Science and Mobile Computing*, 2(3):35–40, 2013.

[17] John Kenney and Ernest Keeping. Linear Regression and Correlation. *Mathematics of statistics*, 1:252–285, 1962.

[18] David Freedman. *Statistical Models: Theory and Practice.* Cambridge University Press, 2009. ISBN 9780521112437.

[19] Harris Drucker, Christopher Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support Vector Regression Machines. In *Advances in Neural Information Processing Systems*, pages 155–161, 1997.

[20] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Spartan Books, 1961.

[21] Peter Whitle. *Hypothesis Testing in Time Series Analysis*, volume 4. Almqvist & Wiksells, 1951.

[22] Felix Gers, Nicol Schraudolph, and Jürgen Schmidhuber. Learning Precise Timing with LSTM Recurrent Networks. *Journal of Machine Learning Research*, 3(Aug):115–143, 2002.

[23] Mohamed Aly. Survey on Multiclass Classification Methods. *International Journal of Computer Science and Information Technologies*, 4(4):572–576, 2005.

[24] Leo Breiman, Jerome Friedman, Charles Stone, and Richard Olshen. *Classification and Regression Trees.* CRC press, 1984.

[25] Tin Kam Ho. Random Decision Forests. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282. IEEE, 1995.

[26] Corinna Cortes and Vladimir Vapnik. Support-vector Networks. *Machine learning*, 20(3):273–297, 1995.

[27] James MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 14, pages 281–297, 1967.

[28] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, volume 96, pages 226–231, 1996.

[29] Teuvo Kohonen. Self-organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, 43(1):59–69, 1982.

[30] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. The KDD Process for Extracting Useful Knowledge from Volumes of Data. *Communications of the ACM*, 39(11):27–34, 1996.

[31] Randall Matignon. *Data Mining using SAS Enterprise Miner*. John Wiley & Sons, 2007. ISBN 9780470149010. doi: 10.1002/9780470171431.

[32] Rüdiger Wirth and Jochen Hipp. CRISP-DM: Towards a Standard Process Model for Data Mining. In *Proceedings of the 4th international Conference on the Practical Applications of Knowledge Discovery and Data Mining*, pages 29–39, 2000.

[33] Ana Azevedo and Manuel Filipe Santos. KDD, SEMMA and CRISP-DM: a Parallel Overview. In *IADIS European Conference on Data Mining*, volume 8, pages 182–185, 2008.

[34] Michael Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.

[35] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[36] Nanette Boden, Danny Cohen, Robert Felderman, Alan Kulawik, Charles Seitz, Jakov Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[37] Gregory Pfister. An Introduction to the Infiniband Architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[38] Mark Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith Underwood, and Robert Zak. Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *Proceedings of the 23rd Annual Symposium on High-Performance Interconnects (HOTI)*, pages 1–9. IEEE, 2015.

[39] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[40] José Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touriño. Locality-aware Automatic Parallelization for GPGPU with OpenHMPP Directives. *International Journal of Parallel Programming*, 44 (3):620–643, 2016.

[41] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC—First Experiences with Real-world Applications. In *Proceedings of Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

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

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

[44] Alexander Heinecke, Michael Klemm, and Hans-Joachim Bungartz. From GPGPU to Many-core: NVIDIA Fermi and Intel Many-integrated-core-architecture. *Computing in Science & Engineering*, 14(2):78–83, 2012.

[45] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU Acceleration Platform. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 8403, 2012. doi: 10.1117/12.921122.

[46] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, and Matthieu Devin. Tensorflow: Large-scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[47] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, Manish Amde, and Sean Owen. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[48] Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. GPU Acceleration for Support Vector Machines. In *Proceedings of the 12th Internation Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011)*, 2011.

[49] Edward Chang, Kaihua Zhu, Hao Wang, Hongjie Bai, Jian Li, Zhihuan Qiu, and Hang Cui. Parallelizing Support Vector Machines on Distributed Computers. In *Advances in Neural Information Processing Systems*, pages 257–264, 2008.

[50] Matthias Richerzhagen. Optimization and Parallelization of PiSvM (*Optimierung der Parallelisierung von PiSvM*). Bachelor Thesis, Fachhochschule Aachen, 2014. Original in German.

[51] Andreas Buyer and Wulf Schubert. Calculation the Spacing of Discontinuities from 3D Point Clouds. *Procedia Engineering*, 191:270–278, 2017.

[52] Mostofa Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. A New Scalable Parallel DBSCAN Algorithm using the Disjoint-set Data Dtructure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 62. IEEE Computer Society Press, 2012.

[53] Geoffrey Fox, Mark Johnson, Gregory Lyzenga, Steve Otto, John Salmon, and David Walker. *Solving Problems on Concurrent Processors: General Techniques and Regular Problems*. Prentice-Hall, Inc., 1988.

[54] Henrike Scholz. Intoxicated Fly Brains: Neurons Mediating Ethanol-induced Behaviors. *Journal of Neurogenetics*, 23(1-2):111–119, 2009.

[55] Aylin Rodan and Adrian Rothenfluh. The Genetics of Behavioral Alcohol Responses in Drosophila. *International Review of Neurobiology*, 91:25–51, 2010.

[56] David Lowe. Distinctive Image Features from Scale-invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[57] Alexander de Brebisson and Giovanni Montana. Deep Neural Networks for Anatomical Brain Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 20–28, 2015.

[58] Yuhui Zheng, Byeungwoo Jeon, Danhua Xu, QM Wu, and Hui Zhang. Image Segmentation by Generalized Hierarchical Fuzzy c-means Algorithm. *Journal of Intelligent & Fuzzy Systems*, 28(2):961–973, 2015.

[59] Kjell Nordberg, Helena L Filipsson, Mikael Gustafsson, Rex Harland, and Per Roos. Climate, Hydrographic Variations and Marine Benthic Hypoxia in Koljö Fjord, Sweden. *Journal of Sea Research*, 46(3):187–200, 2001.

[60] Lina Perelman, Jonathan Arad, Mashor Housh, and Avi Ostfeld. Event Detection in Water Distribution Systems from Multivariate Water Quality Time Series. *Environmental Science & Technology*, 46(15):8212–8219, 2012.

[61] Haifeng Zhao, Dibo Hou, Pingjie Huang, and Guangxin Zhang. Water Quality Event Detection in Drinking Water Network. *Water, Air, & Soil Pollution*, 225(11):1–15, 2014.

[62] Gonzalor Arce and Michael McLoughlin. Theoretical Analysis of the max/median Filter. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(1):60–69, 1987.

[63] Henrik Madsen. *Time Series Analysis*. CRC Press, 2007.

[64] David Powers. Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[65] James W Perry, Allen Kent, and Madeline M Berry. Machine Literature Searching x. Machine Language; Factors Underlying its Design and Development. *Journal of the Association for Information Science and Technology*, 6 (4):242–254, 1955.

[66] Anne H Schistad Solberg, Torfinn Taxt, and Anil K Jain. A Markov Random Field Model for Classification of Multisource Satellite Imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 34(1):100–113, 1996.

[67] Gabriele Cavallaro, Morris Riedel, Matthias Richerzhagen, Jón Atli Benediktsson, and Antonio Plaza. On Understanding Big Data Impacts in Remotely Sensed Image Classification using Support Vector Machine Methods. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8(10):4634–4646, 2015.

[68] Adriana Romero, Carlo Gatta, and Gustavo Camps-Valls. Unsupervised Deep Feature Extraction of Hyperspectral Images. In *Proceedings of the 6th Workshop on Hyperspectral Image Signals Processing: Evolution in Remote Sensensing (WHISPERS)*, 2014.

[69] Yushi Chen, Hanlu Jiang, Chunyang Li, Xiuping Jia, and Pedram Ghamisi. Deep Feature Extraction and Classification of Hyperspectral Images based on Convolutional Neural Networks. *IEEE Transactions on Geoscience and Remote Sensing*, 54(10):6232–6251, 2016.

[70] Gabriele Cavallaro, Mauro Dalla Mura, Jón Atli Benediktsson, and Lorenzo Bruzzone. Extended Self-dual Attribute Profiles for the Classification of Hyperspectral Images. *IEEE Geoscience and Remote Sensing Letters*, 12(8): 1690–1694, 2015.

[71] Philippe Salembier, Albert Oliveras, and Luis Garrido. Antiextensive Connected Operators for Image and Sequence Processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.

[72] Edwin Carlinet and Thierry Géraud. A Comparison of Many Max-tree Computation Algorithms. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 73–85. Springer, 2013.

[73] Michael Dillencourt, Hanan Samet, and Markku Tamminen. A General Approach to Connected-component Labeling for Arbitrary Image Representations. *Journal of the ACM (JACM)*, 39(2):253–280, 1992.

[74] Neil Immerman and Eric Lander. Describing Graphs: A First-order Approach to Graph Canonization. *Complexity Theory Retrospective*, 1, 1990.

[75] Patrick Flick, Chirag Jain, Tony Pan, and Srinivas Aluru. A Parallel Connectivity Algorithm for de Bruijn Graphs in Metagenomic Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 15. ACM, 2015.

[76] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An Overview of the HDF5 Technology Suite and its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

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

[78] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2017. URL `https://www.R-project.org`. [Online, Accessed: 2017-08-24 17:19].

[79] Jeff Bezanzon, Stefan Karpinski, Viral Shah, and Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing. In *Lang.NEXT*, 2012. URL `http://julialang.org/images/lang.next.pdf`.

[80] Bill Hoffman, David Cole, and John Vines. Software Process for Rapid Development of HPC Software using CMake. In *Proceedings on DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC)*, pages 378–382. IEEE, 2009.

[81] Arpan Sen. A Quick Introduction to the Google C++ Testing Framework. *IBM DeveloperWorks*, page 20, 2010.

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

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

[84] Eli Bressert. *SciPy and NumPy: an Overview for Developers.* O'Reilly Media, 2012.

[85] Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, pages 1–9, 2011.

[86] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian Witten. The WEKA Data Mining Software: an Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[87] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[88] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*, 2011.

[89] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015. [Online, Accessed: 2017-08-26 12:05].

[90] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *Advances Neural Information Processing Systems*, 2015.

[91] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[92] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a Modular Machine Learning Software Library. Technical report, IDIAP, 2002.

[93] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[94] Ryan Curtin, James Cline, Neil Slagle, William March, Parikshit Ram, Nishant Mehta, and Alexander Gray. MLPACK: A scalable C++ Machine Learning Library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[95] Chris Thornton, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD, International Conference on Knowledge Discovery and Data Mining*, pages 847–855. ACM, 2013.

[96] James Bergstra, Dan Yamins, and David Cox. Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20, 2013.

[97] Helmut Neukirchen. Survey and Performance Evaluation of DBSCAN Spatial Clustering Implementations for Big Data and High-Performance Computing Paradigms. Technical report, University of Iceland, 2016.

[98] Hans Graf, Eric Cosatto, Leon Bottou, Igor Dourdanovic, and Vladimir Vapnik. Parallel Support Vector Machines: The Cascade SVM. In *Advances in Neural Information Processing Systems*, pages 521–528, 2005.

[99] Hai-fang Zhou, Yan-huang Jiang, and Xue-jun Yang. An Improved Parallel Watershed Algorithm for Distributed Memory System. In *Proceedings of 5th International Conference on Algorithms and Architectures for Parallel Processing*, pages 310–313. IEEE, 2002.

[100] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[101] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[102] Anil Jain and Richard Dubes. *Algorithms for Clustering Data.* Prentice-Hall, 1988.

[103] Brendan O'Donoghue, Remi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. Combining Policy Gradient and Q-learning. 2016.

[104] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[105] Brendan O'Donoghue, Remi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. PGQ: Combining Policy Gradient and Q-learning. *arXiv preprint arXiv:1611.01626*, 2016.

[106] Sonal Kumari, Poonam Goyal, Ankit Sood, Dhruv Kumar, Sundar Balasubramaniam, and Navneet Goyal. Exact, Fast and Scalable Parallel DBSCAN for Commodity Platforms. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 14. ACM, 2017.

[107] Son Mai, Ira Assent, and Martin Storgaard. AnyDBC: an Efficient Anytime Density-based Clustering Algorithm for very Large Complex Datasets. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1025–1034. ACM, 2016.

[108] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD, International Conference on Management of Data*, volume 28, pages 49–60. ACM, 1999.

[109] Ugo Moschini, Arnold Meijster, and Michael Wilkinson. A Hybrid Shared-Memory Parallel Max-Tree Algorithm for Extreme Dynamic-Range Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[110] Pascal Monasse and Frederic Guichard. Fast Computation of a Contrast-invariant Image Representation. *IEEE Transactions on Image Processing*, 9 (5):860–872, 2000.

[111] Gabriele Cavallaro, Mauro Dalla Mura, Jón Atli Benediktsson, and Antonio Plaza. Remote Sensing Image Classification using Attribute Filters Defined over the Tree of Shapes. *IEEE Transactions on Geoscience and Remote Sensing*, 54(7):3899–3911, 2016.

[112] European Commision CORDIS. DEEP-EST. `http://cordis.europa.eu/project/rcn/210094_en.html`, 2017. [Online, Accessed: 2017-08-31 16:21].

[113] Staff writers. Helmholtz invests 17 million Euro into information and data science (*Helmholtz investiert 17 Millionen Euro in den Bereich Information und Data Science*), August 2017. URL `https://idw-online.de/de/news677496`. Original in German, [Online, Accessed: 2017-08-31 16:09].

Appendices

Appendix A

Paper I

Procedia Computer Science
Volume 51, 2015, Pages 2188–2197
ICCS 2015 International Conference On Computational Science

On Scalable Data Mining Techniques for Earth Science

Markus Götz[1,2], Matthias Richerzhagen[1], Christian Bodenstein[1,2], Gabriele Cavallaro[2], Philipp Glock[1], Morris Riedel[1,2], and Jón Atli Benediktsson[2]

[1] Jülich Supercomputing Center (JSC), Forschungszentrum Jülich, Jülich, Germany
{m.goetz, m.richerzhagen, c.bodenstein, p.glock, m.riedel}@fz-juelich.de
[2] School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland
{cavallaro, benedikt}@hi.is

Abstract
One of the observations made in earth data science is the massive increase of data volume (e.g, higher resolution measurements) and dimensionality (e.g. hyper-spectral bands). Traditional data mining tools (Matlab, R, etc.) are becoming redundant in the analysis of these datasets, as they are unable to process or even load the data. Parallel and scalable techniques, though, bear the potential to overcome these limitations. In this contribution we therefore evaluate said techniques in a High Performance Computing (HPC) environment on the basis of two earth science case studies: (a) Density-based Spatial Clustering of Applications with Noise (DBSCAN) for automated outlier detection and noise reduction in a 3D point cloud and (b) land cover type classification using multi-class Support Vector Machines (SVMs) in multi-spectral satellite images. The paper compares implementations of the algorithms in traditional data mining tools with HPC realizations and *'big data'* technology stacks. Our analysis reveals that a wide variety of them are not yet suited to deal with the coming challenges of data mining tasks in earth sciences.

Keywords: Data Mining, Machine Learning, HPC, DBSCAN, SVM, MPI

1 Introduction

The enormous increase in variety, velocity, and volume of spatio-temporal earth science datasets raises data management concerns for organizations storing and preserving the data. There are a number of concrete examples in view of earth science data repositories. One of them is the PANGAEA earth science data collection [7] that has reached, at the time of writing, approximately nine billion data items within around 350,000 datasets. These consist of increasingly large and complex data which often represent long lasting time series of measurement devices from a multitude of different sensors. Another example are the large data collection of remote sensing images [2], taken by airborne sensors or satellites observing, measuring, and recording the radiation reflected or emitted by the Earth and its environment. The *'big data'* challenges in this case concern the improvements of remote sensing capabilities and the availability of

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015
© The Authors. Published by Elsevier B.V.

doi:10.1016/j.procs.2015.05.494

remotely sensed images with high geometrical resolution (e.g. WorldView-2 0.5m) or more detailed spectral information in terms of precision, frequency, or complexity (e.g. AVIRIS 224 spectral channels).

Researchers, however, do not only face issues with the management of the data, but also when attempting to analyze, interpret, or understand them. Due to their *'big data'* properties, they are difficult to process, making it a problem worth investigating. One of the known quotes in that context is: *'Big data is data that becomes large enough that it cannot be processed using conventional methods* [19]. While this definition seems vague, we would like to take the viewpoint in this contribution that these conventional methods span the wide variety of serial tools existing to analyze earth science datasets (e.g. R, Matlab, Weka, etc.). While trying to use some of the already mentioned databases, we have been unable to fully analyze or even just load them into these systems because of memory restriction or highly inefficient implementation.

Handling processing intensive, large datasets, though, is not an entirely new problem, especially when considering computation environments driven by High Performance Computing (HPC). We would like to find out what benefits of these environments are applicable to the analysis of large quantities of earth science datasets. In the foreseeable future, and in order to better understand many of the climate issues we face today, we even require a systematic approach of combining and studying earth science datasets derived both from observations and model simulations often running in HPC environments. Therefore, this contribution aims to provide a deeper understanding about two concrete case studies that are able to take advantage of parallel and scalable data mining techniques of large quantities of earth science data. Throughout the paper, and as part of these studies, we provide pieces of information about strengths and weaknesses of various *'big data'* technologies and comparing approaches to HPC environments and their strong capabilities that are also applicable to statistical earth science data mining tasks.

The first case study covers the known territory of finding outliers in data using clustering algorithms. While many different clustering algorithms are potentially applicable [16], we focus here on the Density-based Spatial Clustering of Applications with Noise [8] known as DBSCAN. One example of a relevant data source is a marine observatory (i.e. Koljoefjord observatory) that delivers continuous measurements over a long period of time. We study the use of DBSCAN in order to find out if we are able to detect events or outliers that can also support the data quality management. A scalable and robust automated outlier detection system that is able to cope with large volumes of data within PANGAEA is therefore required. During our study we were unable to find a suitable implementation of the algorithm allowing us to perform the described analysis. For this reason we had to develop our own scalable solution of a parallel DBSCAN delaying the actual analysis of the PANGAEA Koljoefjord observations. We will demonstrate the capabilities of the software on the basis of a simpler, but visually tangible, earth science outlier use case, the denoising of a 3D point cloud.

Our second case study deals with land cover mass classification in remote sensing image datasets. The classification problem aims to categorize all pixels in a digital image into meaningful classes of land cover types in a particular scene. In order to obtain a satisfactory level of detection accuracy, we perform a detailed physical analysis by exploiting the availability of high spatial resolution images. Hence, we consider attribute filters, flexible operators that can transform an image according to many different attributes (e.g. geometrical, textural, and spectral) as further optimization technique [3]. One often used classification technique in the field of remote sensing are Support Vector Machines (SVMs) [6] and their kernel methods (e.g. radial basis function). The contribution of this paper in the context of classification is the exploration of according SVM frameworks.

This paper is structured as follows. After the introduction into the problem domain in Section 1, key requirements for evaluating parallel and scalable tools for the analysis of scientific and engineering problems are given in Section 2. Section 3 offers a thorough survey of related work in the field of parallel and scalable data mining tools with a particular focus on parallel DBSCAN and SVM implementations. Section 4 highlights selected technical details on two parallel implementations we use to process the aforementioned earth science datasets. The paper ends with some concluding remarks.

2 Requirements

Technical and algorithmic solutions for the aforementioned case studies need to satisfy a number of key requirements listed in this section. Satisfying those requirements and overcoming scalability constraints given by the available datasets thus represent one of our major motivation. There is a wide variety of traditional tools available and an increasing number of more recent *'big data stacks'* that claim to support parallel and scalable data mining in one way or the other. We picked the following (without considering commercial tools) for our deeper analysis: (i) Weka, (ii) R, (iii) Matlab, (iv) Octave, (v) Apache Mahout, (vi) MLlib/Apache Spark, (vii) scikit-learn, and individual implementations for the specific algorithm in question. Those tools are analysed in terms of three different criteria *'(a) open and free availablility'*, *'(b) technical feasibility'*, and *'(c) suitability of algorithms'* in the light of the motivating case studies. Despite the fact that there is an ever increasing number of rather new *'big data stacks'*, our work is also motivated by exploring whether traditional HPC environments play a role in mining *'big data'*. Although the HPC environments are typically driven by demands of the *'simulation sciences'*, based on efficient numerical methods and known physical laws, some computational science applications raise similar requirements to the processing environments as it is the case for data mining tasks. In this contribution we therefore focus on comparing solutions based on the following five key capabilities we consider important when analyzing technical solutions:

- R1 Parallel file systems and large storage capacity
- R2 Scalable standard data formats that take advantage of parallel I/O
- R3 Standard communication protocols
- R4 Fair scheduling tools and policies to enable the work on clusters for several users at a time
- R5 Open source tools and open referencable data to enable reproducability of findings

3 Related Work

This section is structured along the key requirements and framework collection introduced in section 2.

3.1 Survey of DBSCAN Clustering Tools

Clustering is an established data mining method that can be best explained by dividing data into subgroups of similar items, whereby each member within such a found group is similar to all the others and different from the members of other groups. The similarity is defined by a problem-specific metric like for instance the euclidean distance between data items. Cluster

analysis belongs to the so-called unsupervised learning methods since the input data items $X_i =< x_1, \ldots, x_d >\in \mathbb{R}^d$ with $i = 1 \ldots N$ are given, but no desired results or ground truth to be learned from[1].

In this survey we focus driven by our outlier use case on the DBSCAN clustering algorithm. The literature offers several publications describing parallelization approaches, e.g. based on the map-reduce paradigm [12], using distribute dR-Trees [5], or more traditional variants in the context of databases [13]. However, except of one, none of these offer concrete implementations or point to respective source code repositories. We could therefore only include PDSDBSCAN-D [15] in our survey despite the list of frameworks introduced in the method section.

Technology	Platform Approach	Supports DBSCAN	Parallelization	Stable
Weka	Java	yes	no	yes
R	R	yes	no	yes
Matlab	Matlab	no	no	yes
Octave	Octave	no	no	yes
Apache Mahout	Java, Hadoop	no	yes	yes
MLlib/Apache Spark	Java, Spark, Hadoop	no	yes	yes
scikit-learn	Python	yes	no	yes
PDSDBSCAN-D	C++, MPI, OpenMP	yes	yes	no
HPDBSCAN	C++, MPI, OpenMP	yes	yes	yes

Table 1: Overview of open and freely available DBSCAN clustering tools

Table 1 shows the results of our survey. The new *'big data'* technology stacks Apache Mahout and Spark do not offer an implementation of DBSCAN to begin with. Interestingly enough the same is true for the well-known data mining tools Matlab and Octave that do not support this kind of analysis. In contrast to that Weka, R and scikit-learn have packages that allow to cluster data using DBSCAN. However, all of them are not parallelized and therefore do not have the ability to scale to modern HPC systems. Only the one remaining implementation satisfies the *'(a) open and free availability'* criteria as well as the *'(c) suitability of algorithms'* criteria – PDSDBSCAN-D. A deeper analysis of the C++ code based on MPI or OpenMP reveals drawbacks in terms of *'(b) technical feasibility'* in terms of scalability and speedup while performing tests with real world data [9]. In order to satisfy our demanding earth science outlier case study we have implemented a more efficient parallel version of DBSCAN that we named HPDBSCAN. Our approach differs from the PDSDBSCAN-D implementation in several ways such as a smart preprocessing into spatial cells as well as density-based chunking to load balance the local computation.

3.2 Survey of SVM Classification Tools

Classification is the problem of identifying the membership of data items into a set of subgroups called classes. In contrast to clustering, the statistical model for this is learned from known observations or samples of class memberships. Classification therefore belongs to the supervised learning methods since input data items $X_i =< x_1, \ldots, x_d >\in \mathbb{R}^d$ with $i = 1 \ldots N$ are given in combination *'with supervising output'* data y_i. In our survey we focus on the support vector machine (SVM) classification algorithm, including its kernel methods allowing to classify non-linearly seperable data.

[1]The variable d equals the number of dimensions or features and N the number of different samples

Technology	Platform Approach	Multiclass	Supported Kernels	Parallelization	Stable
Weka	Java	yes	linear, rbf, polynomial, sigmoid	no	yes
R (kernlab)	R	yes	linear, rbf, polynomial, sigmoid	no	yes
Matlab	Matlab	yes	linear, rbf, polynomial, sigmoid	no	yes
Octave	Octave	yes	linear, rbf, polynomial, sigmoid	no	yes
Apache Mahout	Java, Hadoop	-	-	-	-
MLlib/Apache Spark	Java, Spark, Hadoop	no	linear	yes	yes
scikit-learn	Python	yes	linear, rbf, polynomial, sigmoid	no	yes
libSVM	C, Java	yes	linear, rbf, polynomial, sigmoid	no	yes
Twister/ParallelSVM	Java, Twister, Hadoop	no	linear, rbf, polynomial, sigmoid	yes	no
pSVM	C, MPI	no	linear, rbf, polynomial	yes	no
GPU LibSVM	CUDA	yes	linear, rbf, polynomial, sigmoid	yes (rbf)	yes
πSVM	C, MPI	yes	linear, rbf, polynomial, sigmoid	yes	yes

Table 2: Overview of open and freely available SVM classification tools

As shown in Table 2, there is a wide variety of candidate solutions tools available that vary in terms of their platform approach. In terms of *'(c) suitability of algorithms'*, we have found only four of the different implementations to be useful. The reasoning is as follows. Weka, R, Matlab, Octave and scikit-learn, as high-level languages also attractive to non technically savy scientists, wrap libSVM as a base implementation. Due to the lack of parallelization of the latter, however, it is not suitable for our earth science data mining problem. MLlib of Apache Spark seemingly overcomes this by reimplementing SVMs using the scalable and parallel Spark core, but a deeper analysis show that it only supports linear SVMs. Again, this is not applicable for our earth science data mining use case as we have non linearly separable classes. The remaining four satisfy basically the *'(c) suitability of algorithms'* requirement, but a deeper analysis reveals further drawbacks in terms of *(b) technical feasibility.* The implementation of pSVM [20] as well as Twister [18] are unstable beta releases that are not meant to be used in production yet. While using Twister for example we found dependencies to messaging systems, scheduling issues, and other problems related to the lack of features that have made an analysis of our data challenging. Finally, there are only two parallel implementations left. The GPU LibSVM bears lots of potential for future use, but due to its dependencies to the proprietary Compute Unified Device Architecture [14] technology stack, we have concerns regarding our *'(a) openly and free available'* criteria. For this reason, we have decided to test πSVM 1.2 (and indirectly the recent πSVM1.3) with our data. Overall the performance has been acceptable, but some scalability issues, have lead us to optimize it further, which we will present in section 4.2.

4 Parallel and Scalable Methods

This section gives insights about our parallel and scalable methods and their optimization structured alongside the approaches raised from the two scientific case studies introduced in Section 1. Beside technical details, we also discuss in this section the HPC environment benefits that satisfy the majority of our key requirements defined in Section 2 (i.e. R1-R5).

4.1 Parallel and Scalable Clustering with HPDBSCAN

DBSCAN is a density-based clustering algorithm and its principal idea is to find cluster cores in a data set and subsequently expand these recursively. Thereby, a cluster core is defined as region that contains at least a parametric number *minPoints* of neighboring point within a given search radius *epsilon* (ε) and with respect to a distance function dist. For each of

these neighboring points the same criterion is reapplied in order to extend the found cluster. Highly parallelizable DBSCAN (HPDBSCAN) is our parallel implementation of DBSCAN. It overcomes the inherent sequential processing step of the recursive expansion through the following three major techniques.

Spatial Indexing Structure Common techniques for indexing data points are dR- or kd-trees [17] as they have been used in other parallel DBSCAN research work. These significantly speeds up region queries, especially also for dynamic search radius. In DBSCAN, however, we have a constant search radius equaling to epsilon. Based on this fact, we have chosen a different spatial indexing approach for HPDBSCAN, which uses a sorted, regular n-dimensional hypergrid indexed by hash tables that overlays the n-dimensional data space (cf. Figure 1). Similarily to the *'big data'* array databases rasdaman or SciDB [1], we can answer queries based on this approach faster, in armortized $\mathcal{O}(1)$ complexity and can cash and preload queries.

Quadratic Split Heuristic Instead of distributing the data items in equal-sized chunks to all processor, like introduced in previous DBSCAN research works, we use a heuristic to achieve better load balancing, especially for spatially skewed data. Since the computation time of DBSCAN scales in a quadratic manner with respect to the point density, we calculate a score value for each cell of the hypergrid, which take advantage of this scaling behavior. The scores allows us to find exact splittings of the hypergrid, such that no communication is required during the parallel computation step. At the same time, we able to balance the workload evenly among processors.

Differential Merging Scheme HPDBSCAN is able to combine the sub-results of the parallel clustering into one common view. Therefore, it exchanges the direct bordering region halos of the hypergrid splits, finds deviating cluster labels, and rewrites them accordingly. In the merging step, we store conflicting labels in dictionaries, so that we can apply relabeling of points in parallel afterwards. In contrast, to other parallel DBSCAN implementations such as PDSDBSCAN-D [15] there is no additional communication or clustering needed during the re-labeling, which increases the computational performance tremendously.

We have realized HPDBSCAN as a MPI and OpenMP hybrid in C++ that is usable as a standalone command line interface (CLI) tool as well as a shared library that can be wrapped or linked to bigger applications if needed. In case of the CLI program, data points are provided in form of Hierarchical Data Format (HDF5) files [11], in which the clustering results are also written back. These clustering results per point are either the identity of a specific cluster or an outlier mark that in particular is useful for our earth science case study.

In order to demonstrate its scalability and its outlier detection potential with large volumes of data, we are filtering a 3D point cloud of the old town of Bremen for outliers, such as false readings or points which capture the inside of a building. The points cloud contains over 81 million individual points [10], clustering these sequentially would require days. Using HPDBSCANand the supercomputer JUDGE at the research center Jlich, we can cut this time down to minutes. Figure 2 shows an example of the point cloud in a version that has been clustered using HPDBSCANwith the noise points still in, but colored in red. These can be automatically removed, allowing us to start our next analysis steps. Our implementation takes advantage of (R1) parallel file systems and large storage capacity existing in many HPC environments. We a are able to fully exploit (R2) scalable standard data formats that take advantage of parallel/IO due to the adoption of HDF5. The code is us sing (R3) standard

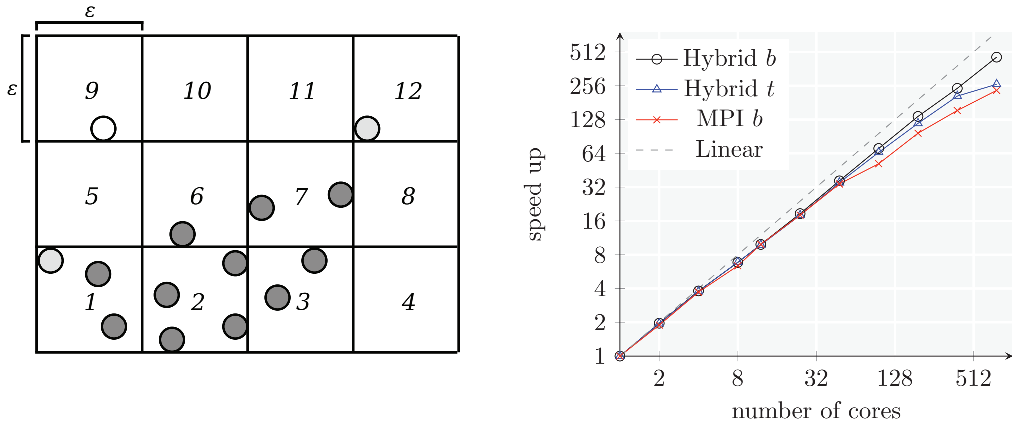


Figure 1: Left: Schematic illustration of the HPDBSCAN overlayed hypergrid in a two dimensional problem. The points are indexed by hash table pointing to each grid cell that has a side length of ε; Right: Speed-up and scaling of HPDBSCAN.

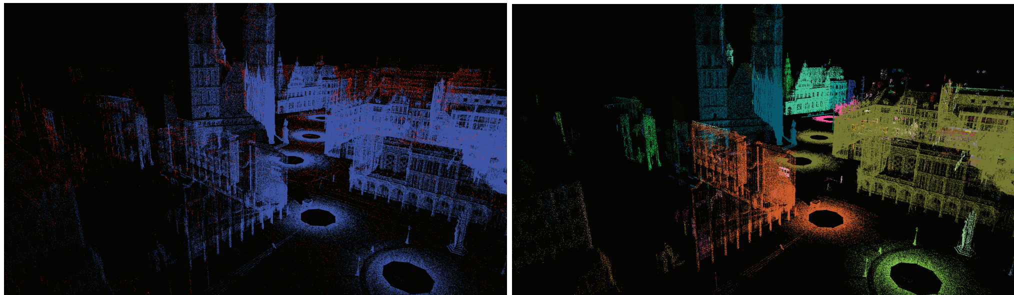


Figure 2: Left: Result ts of HPDBSCAN including noise reduction; Right: Identified clusters that are illustrated using different colors.

communication protocols and enables the use of (R4) fair and stable scheduling tools and policies to enable the work on clusters for several users at the same time. Given that HPDBSCAN is open source [9] it belongs to the (R5) open source tools and open referencable data in PANGAEA [7] and Bremen [10], respectively, enables reproducibility of findings.

4.2 Parallel and Scalable Classification with πSVM

In this section we present our earth science classification use case in remote sensing preceded by some technical details of the used SVM software stack. As introduced in section 3.2 we have based our parallel SVM implementation on πSVM version 1.2. As of late 2014 version 1.2 is outdated, though, as its successor version 1.3 has been released. For time reasons we were unable to port our changes to the new version, but our outlined performance advancements are still valid, due to the fact that πSVM 1.3 does not optimized the parallel code sections.

As shown in Figure 3, our optimized πSVM implementation scales better in constant sized

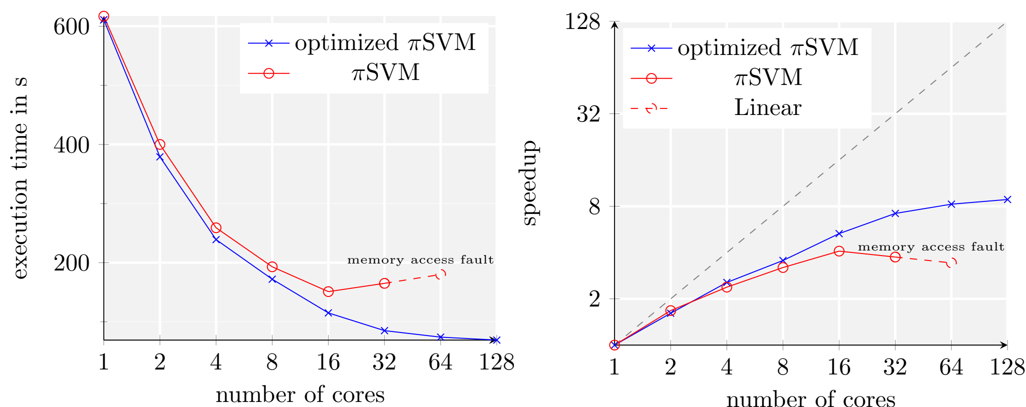


Figure 3: Comparison between the original πSVM version and its optimized version (including memory access fault). Left: Execution time as a function of doubling processing cores count. Right: Speedup values of constant-sized problem.

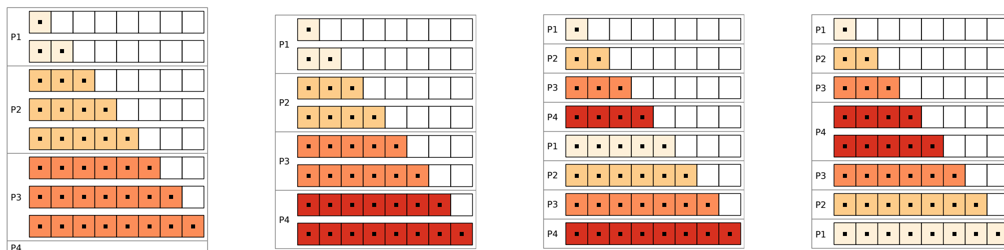


Figure 4: Parallel memory access patterns in a triangular matrix using four processing cores. From left to right: initially implemented in πSVM, naive equal chunking, implemented in our optimized πSVM tool, theoretically ideal pattern.

problems compared to the original πSVM implementation. While we still work on achieving better scalability (e.g. consideration of a hybrid implementation using OpenMP), the code has proven to work with our earth science classification case study. One of the major performance gains has been achieved through the usage of more suitable MPI collective operations. For example, in some source code sections a simple for loop was used to perform an `MPI_Bcast()` numerous times to inform other processors about training updates. Instead we replaced them with singular `MPI_Allgather()` calls reducing it from linear to constant time and memory complexity. A couple of other optimization strategies have been implemented. One of them is an improved matrix distribution pattern during the SVM training stage to balance the computational load, schematically illustrated in figure 4. The unoptimized πSVM code distributes them as depicted in the left picture, leaving processing cores idle. We have improved this distribution pattern step-wise from a naive equal-sized chunking, to an skip-row approach and finally to a theoretically ideal on the right. In tests, however, we have found that the skip-row approach outperforms all other patterns, due its simple implementation using one MPI collective operation, in contrast to the complex pattern required for the theoretically best.

We have used our parallel and scalable πSVM implementation in different classification

studies in the field of remote sensing, e.g. on remote sensing image classification of the city Rome that we published at a remote sensing domain-specific conference IGARSS 2014 [4]. Since then we have shown of the scalability of the code by investigating more complex problems, such asthe Indian Pines dataset with over 50 classes. The dataset is openly available through EUDATs B2SHARE services [4] which enables reproducibility of the papers findings.

Our optimized πSVM implementation takes advantage of (R1) parallel file systems and large storage capacity existing in many HPC environments and is based on MPI thus using (R3) standard communication protocols. The code can be used with typical HPC batch schedulers enabling the use of (R4) fair and stable scheduling tools and policies to enable the work on clusters for several users at the same time. Given that πSVM is open source as well as our optimized version the (R5) open source tools and open referencable data enables reproducibility of findings. Potential for future work is to enable the use of parallel I/O that is currently not supported (cf. R2).

5 Conclusion

One of the major findings of this paper is the need for parallel and scalable data mining tools in earth sciences, as has been demonstrated by the means of the two case studies. While we face these challenges in machine learning already now, the expected increase of *'big data'* in science in the coming years, will amplify this need even more. With our two parallel data mining tools, HPDBSCAN and πSVM, we start to tackle this problem, as both of them show significant improvements compared to their serial counterparts. However, they still only utilize a moderate number (several dozens to hundreds) of cores and do not yet fully exploit the capabilities of a large HPC supercomputer.

Given the page restriction, our study also did not describe all potential parallel optimization points entirely in detail. One straightforward example is the use of cross-validation for model selection as it is implemented in πSVM. Another example is the so-called grid search technique, which is a highly computational intensive, and that typically aims to explore the right combination of machine learning algorithm parameters (i.e. here for the SVM and its rbf kernel). Hence, easy to implement parallelizations of this processes should enable additional massive speedups compared to serial implementations making thus the use of parallel and scalable methods even more feasible.

Future work beyond using the ever increasing amounts of datasets faced in earth sciences, is the exploration of in situ analytics towards exascale computing. In other words the exascale simulation (e.g. of a climate model) is running while another part of the machine (e.g. using GPGPUs or accelerators) is performing analytics in situ in order to validate the simulation with real measurement data and to perform statistics on the fly on the created simulated data. This bears the potential to abort costly computation runs early based on the statistical data mining that will take place concurrently to the particular numerical simulation.

Acknowledgments

We would like to express our thanks to the EUDAT European data infrastructure in general and the B2SHARE service support team in particular for providing us with storage space and handles to properly link research data.

References

[1] Peter et al. Baumann. The array database that is not a database: file based array query answering in rasdaman. In *Advances in Spatial and Temporal Databases*, pages 478–483. Springer, 2013.

[2] James B Campbell. *Introduction to remote sensing*. CRC Press, 2002.

[3] Gabriele et al. Cavallaro. A comparison of self-dual attribute profiles based on different filter rules for classification. In *Geoscience and Remote Sensing Symposium (IGARSS), 2014 IEEE International*, pages 1265–1268. IEEE, 2014.

[4] Gabriele et al. Cavallaro. Smart data analytics methods for remote sensing applications. In *Geoscience and Remote Sensing Symposium (IGARSS), 2014 IEEE International*, pages 1405–1408. IEEE, 2014.

[5] Min et al. Chen. Parallel dbscan with priority r-tree. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 508–511. IEEE, 2010.

[6] Corinna et al. Cortes. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[7] Michael et al. Diepenbroek. PANGAEAan information system for environmental sciences. *Computers & Geosciences*, 28(10):1201–1210, 2002.

[8] Martin et al. Ester. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, number 34 in 96, pages 226–231, 1996.

[9] Markus Götz. Clustering - HPDBSCAN. `http://www.fz-juelich.de/ias/jsc/EN/Research/DistributedComputing/DataAnalytics/Clustering/Clustering_node.html`, 2015. [Online; accessed Feburary, 10th 2015; 11:14 CET].

[10] Markus et al. Götz. HPDBSCAN Benchmark test files. `URL:https://b2share.eudat.eu/record/178`, 2015. [Online; Accessed Febuary, 9th 2015; 12:09 CET].

[11] HDF Group. Hierarchical data format version 5. `http://www.hdfgroup.org/HDF5`, 2015. [Online; Accessed Febuary, 10th 2015; 14:17 CET].

[12] Yaobin et al. He. MR-DBSCAN: an efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 473–480. IEEE, 2011.

[13] Eshref et al. Januzaj. Scalable density-based distributed clustering. In *Knowledge Discovery in Databases: PKDD 2004*, pages 231–244. Springer, 2004.

[14] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[15] Md Mostofa Ali et al. Patwary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[16] Sangeeta et al. Rani. Recent techniques of clustering of time series data: A Survey. *Int. J. Comput. Appl*, 52(15):1–9, 2012.

[17] Hanan Samet. *The design and analysis of spatial data structures*, volume 85. Addison-Wesley Reading, MA, 1990.

[18] Zhanquan Sun and Geoffrey Fox. Study on parallel SVM based on MapReduce. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 16–19. Citeseer, 2012.

[19] O Team. Big Data Now: Current Perspectives from OReilly Radar, 2011.

[20] Edward Y Changf Kaihua et al. Zhu. PSVM: Parallelizing Support Vector Machines on Distributed Computers.

Appendix B

Paper II

HPDBSCAN – Highly Parallel DBSCAN

Markus Götz
m.goetz@fz-juelich.de

Christian Bodenstein
c.bodenstein@fz-juelich.de

Morris Riedel
m.riedel@fz-juelich.de

Jülich Supercomputing Center
Leo-Brandt-Straße
52428 Jülich, Germany

University of Iceland
Sæmundargötu 2
101, Reykjavik, Iceland

ABSTRACT

Clustering algorithms in the field of data-mining are used to aggregate similar objects into common groups. One of the best-known of these algorithms is called *DBSCAN*. Its distinct design enables the search for an apriori unknown number of arbitrarily shaped clusters, and at the same time allows to filter out noise. Due to its sequential formulation, the parallelization of *DBSCAN* renders a challenge. In this paper we present a new parallel approach which we call *HPDBSCAN*. It employs three major techniques in order to break the sequentiality, empower workload-balancing as well as speed up neighborhood searches in distributed parallel processing environments *i*) a computation split heuristic for domain decomposition, *ii*) a data index preprocessing step and *iii*) a rule-based cluster merging scheme.

As a proof-of-concept we implemented *HPDBSCAN* as an OpenMP/MPI hybrid application. Using real-world data sets, such as a point cloud from the old town of Bremen, Germany, we demonstrate that our implementation is able to achieve a significant speed-up and scale-up in common HPC setups. Moreover, we compare our approach with previous attempts to parallelize *DBSCAN* showing an order of magnitude improvement in terms of computation time and memory consumption.

Categories and Subject Descriptors

A.2.9 [**General and reference**]: Cross-computing tools and techniques—*Performance*; F.5.8 [**Theory of computation**]: Design and analysis of algorithms—*Parallel algorithms* ; H.3.8 [**Information systems**]: Information systems applications—*Data mining* ; I.2.1 [**Computing methodologies**]: Parallel computing methodologies—*Parallel algorithms*

General Terms

Algorithms, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MLHPC2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4006-9/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2834892.2834894

Keywords

High performance computing, scalable clustering, parallel DBSCAN, HPDBSCAN, OpenMP/MPI hybrid

1. INTRODUCTION

Cluster analysis is a data-mining technique that divides a set of objects into disjoint subgroups, each containing similar items. The resulting partition is called a *clustering*. A clustering algorithm discovers these groups in the data by maximizing a similarity measure within one group of items—or cluster—and by minimizing it between individual clusters. In contrast to supervised learning approaches, such as classification or regression, clustering is an unsupervised learning method. This means that, it tries to find the mentioned structures without any apriori knowledge about the actual ground-truth. Typical fields of application for cluster analysis include sequence analysis in bio-informatics, tissue analysis in neuro-biology, or satellite image segmentation.

Clustering algorithms can be divided into four classes: partitioning-based, hierarchy-based, density-based and grid-based [17]. In this paper we will discuss aspects of the two latter classes. Specifically, we are going to talk about the density-based clustering algorithm *DBSCAN —density-based spatial clustering of applications with noise* [9]—and how to efficiently parallelize it using computationally efficient techniques of grid-based clustering algorithms. Its principal idea is to find dense areas, the cluster cores, and to expand these recursively in order to form clusters. The algorithm's formulation has an inherent sequential control flow dependency at the point of the recursive expansion, making it challenging to parallelize.

In our approach we break the interdependency by adopting core ideas of grid-based clustering algorithms. We overlay the input data with a regular hypergrid, which we use to perform the actual *DBSCAN* clustering. The overlayed grid has two main advantages. Firstly, we can use the grid as a spatial index structure to reduce the search space for neighborhood queries; and secondly, we can separate the entire clustering space along the cell borders and distribute it among all compute nodes. Due to the fact that the cells are constructed in a regular fashion, we can redistribute the data points, by facilitating halo areas, so that there is no intermediate communication required during the parallel computation step.

In spatially skewed datasets regular cell space splits would lead to an imbalanced computational workload, since most of the points would reside in dense cell subspaces assigned to a small number of compute nodes. To mitigate this we propose

a cost heuristic that allows us to identify data-dependent split points on the fly. Finally, the local computation results are merged through a rule-based cluster merging scheme, with linear complexity.

The remainder of this paper is organized as follows. The next section surveys related work. Section 3 describes details of the original *DBSCAN* algorithm. Subsequently, Section 4 discusses our parallelized version of *DBSCAN* by the name of *HPDBSCAN* and shows its algorithmic equivalence. Section 5 presents details of our hybrid OpenMP/MPI implementation. Evaluations are shown in Section 6, where we also present our benchmark method, datasets and the layout of the test environment. We conclude the paper in Section 7 and give an overview of possible future work.

2. RELATED WORK

There are a number of previous research studies dealing with the parallelization of *DBSCAN*. To the best of our knowledge, the first attempt was made by Xiaowei Xu in collaboration with Kriegel et al. [27]. In their approach, single neighborhood search queries are parallelized facilitating a distributed version of the R-Tree—*DBSCAN*'s initial spatial index data structure. They adopt a master-slave model, where the index is built on the master node and the whole data set is split among the slaves according to the bounding rectangles of the index. Subsequently, they merge the local cluster results by reclustering the bordering regions of the splits. Zhou et al. [28] and Arlia et al. [3] present similar approaches, where they accelerate the neighborhood queries by replicating the entire index on each of the slave nodes, assuming the index fits entirely into the main memory. Brecheisen et al. [5] have published a parallel version of *DBSCAN* that approximates the cluster using another clustering algorithm called *OPTICS*. Each of the cluster candidates found in this manner is sent to a slave node in order to filter out the actual from the guessed cluster points. The local results are then merged by the master into one coherent view. This approach, however, fails to scale for big databases, since the pre-filtering has to be done on the master, in main memory. Chen et al. [7] propose another distributed *DBSCAN* algorithm, called *P-DBSCAN* that is based on a priority R-Tree. Unfortunately, the paper does not state how the data is distributed or how the clusters are formed. An in-depth speed and scale-up evaluation is also not performed. A paper by Fu et al. [13] demonstrates the first Map-Reduce implementation of *DBSCAN*. The core idea of this approach is the same as the first parallelization attempt of Xu, that is, to parallelize singular neighborhood queries—this time in form of Map-Tasks. He et al. [19] present another implementation of a parallel *DBSCAN* based on the Map-Reduce paradigm. They are the first to introduce the notion of a cell-based preprocessing step in order to perform a fully distributed clustering without the need to replicate the entire dataset or to communicate inbetween. Finally, Patwary et al. [25] have published research work that shows a parallel *DBSCAN* that scales up to hundreds of cores. Their main contribution is a quick merging algorithm based on a disjoint-set data structure. However, they either need to fit the entire dataset into main memory or need a manual preprocessing step that splits the data within a distributed computing environment.

3. THE *DBSCAN* ALGORITHM

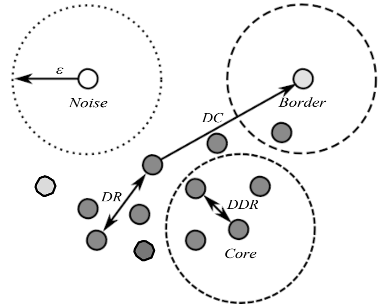


Figure 1: *DBSCAN* **clustering with** $minPoints = 4$

DBSCAN is a density-based clustering algorithm that was published 1996 by Ester et al. [9]. Its principal idea is to find dense areas and to expand these recursively in order to find clusters. A dense region is thereby formed by a point that has within a given search radius ε at least $minPoints$ neighboring points. This dense area is also called the *core* of a cluster. For each of the found neighbor points the density criteria is reapplied and the cluster is consequently expanded. All points that do not form a cluster core and that are not "absorbed" through expansion are regarded as *noise*.

A formal definition of the algorithm is as follows. Let X be the entire dataset of points to be clustered and $p, q \in X$ two arbitrary points of this set. Then the following definitions describe *DBSCAN* with respect to its parameters ε and $minPoints$. Figure 1 illustrates these notions.

Definition 1. Epsilon neighborhood (N_ε)—The epsilon neighborhood N_ε of p denotes all points q of the dataset X, which have a distance $dist(p, q)$ that is less or equal to ε, or formally: $N_\varepsilon(p) = \{q|dist(p, q) < \varepsilon\}$. In practice, the euclidean distance is often used for *dist* making the epsilon-neighborhood of p equal to the geometrically surrounding hypersphere with radius ε.

Definition 2. Core point—p is considered a core point if the epsilon-neighborhood of p contains at least $minPoints$ number of points including itself: $Core(p) = |N_\varepsilon(p)| \geq minPoints$.

Definition 3. Directly density-reachable (DDR)—A point q is directly density-reachable from a point p, if p lies within q's epsilon-neighborhood and p is a core point, i.e., $DDR(p, q) = q \in N_\varepsilon(p) \wedge Core(p)$.

Definition 4. Density-reachable (DR)—A pair of points $p_0 = p$ and $p_n = q$ are called density reachable, if there exists a chain of directly density-reachable points—$\{p_i|0 \leq i \wedge i < n \wedge DDR(p_i, p_{i+1})\}$—linking them with one another.

Definition 5. Border point—Border points are special cluster points that are usually located at the outer edges of a cluster. They do not fulfill the core point criteria but are still included in it due to direct density-reachability. Formally, this can be expressed as $Border(p) = |N_\varepsilon(p)| < minPoints \wedge \exists q : DDR(q, p)$.

Definition 6. Density-connected (DC)—Two points p and q are called density connected, if there is a third point r, such that r can density-reach p and q: $DC(p,q) = \exists r \in X : DR(r,p) \land DR(r,q)$. Note that density-connectivity is a weaker condition than density-reachability. Two border points can be density-connected, even though they are not density-reachable by definition due to not fulfilling the core point criteria.

Definition 7. Cluster—A cluster is a subset of the whole dataset, where each of the points is density-connected to all the other and that contains at least one dense region, or in other words a core point. This can be denoted as $\emptyset \subset C \subseteq X$ with $\forall p,q \in C : DC(p,q)$ and $\exists p \in C : Core(p)$.

Definition 8. Noise—Noise are special points that do not belong to any epsilon-neighborhood, such that $Noise(p) = \neg\exists q : DDR(q, p)$.

Listing 1 sketches pseudo code for a classic *DBSCAN* implementation. Some of the type and function definitions are left out, as their meaning can easily be inferred.

DBSCAN's main properties that distinguish it from more traditional clustering algorithms, such as *k-means* [17] for instance, are: *i*) it can detect arbitrarily shaped clusters that can even protrude into, or surround one another; *ii*) the cluster count does not have to be known apriori, and, *iii*) it has a notion of noise inside the data.

Finding actual values for ε and $minPoints$ is dependent on the clustering problem and its application domain. Ester et al. [9] propose a simple algorithm for estimating ε. The core idea is to determine the "thinnest" cluster area through either visualization or a sorted 4-dist graph, and then choose ε to be equal to that width.

```
1  def DBSCAN(X, eps, minPoints):
2    clusters = list()
3    for p in X:
4      if visited(p):
5        continue
6      markAsVisited(p)
7      Np = query(p, X, eps)
8      if length(Np) < minPoints:
9        markAsNoise(p)
10     else:
11       C = Cluster()
12       add(clusters, C)
13       expand(p, Np, X, C, eps, minPoints)
14   return clusters
15
16 def expand(p, Np, X, C, eps, minPoints):
17   add(p, C)
18   for o in Np:
19     if notVisited(o):
20       markAsVisited(o)
21       No = query(o, X, eps)
22       if length(No) >= minPoints:
23         Np = join(Np, No)
24     if hasNoCluster(o):
25       add(o, C)
```

Listing 1: Classic *DBSCAN* pseudocode

4. *HPDBSCAN*

In this section we present Highly Parallel *DBSCAN*, or in short *HPDBSCAN*. Our approach to parallelize *DBSCAN* consists of four major stages. In the first step the entire dataset is loaded in equal-sized chunks by all processors in parallel. Then, the data is preprocessed. This entails assigning each of the d-dimensional points in the dataset to a virtual, unique spatial cell corresponding to their location within the data space, with respect to the given distance function. This allows us to sort the data points according to their proximity, and to redistribute them to distinct computation units of the parallel computing system. In order to balance the computational load for each of the processing units, we estimate the load using a simple cost heuristic accommodating the grid overlay.

After this division phase, we perform the clustering of the redistributed points in the second step locally on each of the processing units, i.e., we assign a temporary cluster label to each of the data points.

Subsequently, these have to be merged into one global result view in step three. Whenever the temporary label assigned by a processing unit disagrees with the ones in the halo areas of the neighboring processors, we generate cluster relabeling rules.

In the fourth step, the rules are broadcasted and applied locally. Figure 2 shows a schematic overview of the process using the fundamental modeling concepts (FMC) notation [21]. The next sections scrutinizes each of the substeps theoretically.

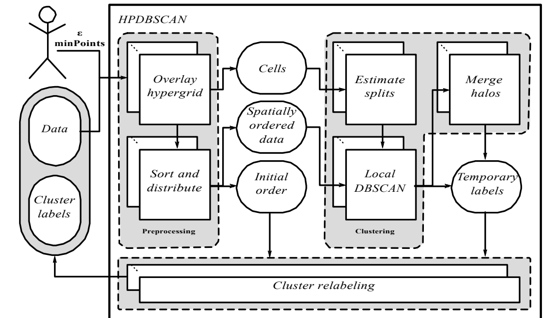


Figure 2: Schematic overview of *HPDBSCAN*

4.1 Grid-based data preprocessing and index

The original *DBSCAN* paper proposes the use of R-trees [4] in order to reduce the neighborhood search complexity from $O(n^2)$ to $O(log(n))$. The construction of the basic R-tree cannot be performed in parallel as it requires the entire dataset to be known. Therefore, other researchers [4, 27] propose to either just replicate the entire dataset, and perform linear neighborhood scans in parallel for each data item, or to use distributed versions of the R- or k-d-trees. However, He et al. [19] point out that these approaches do not scale in terms of memory consumption or communication cost with respect to large datasets and number of parallel processors.

Therefore, we have selected a far more scalable approach for *HPDBSCAN* that is based on grid-based clustering algorithms like, e.g., STING [17], and common spatial problem in HPC, like for example HACC in particle physics [16]. Its core idea is that the d-dimensional bounding box of the entire dataset, with respect to *dist*, is overlayed by a regular, non-overlapping hypergrid structure, which is then decomposed into subspaces by splitting the grid along the grid cell

boundaries. Each of the resulting subspaces is then exclusively assigned to a parallel processor that is responsible for computing the local clustering. In order to be able to do so in a scalable fashion, all the data points within a particular subspace should be in the local memory of the respective parallel processor, so that communication overhead is avoided. However, in most cases the data points will be distributed in arbitrary order within the dataset. Therefore, the data has to be indexed first and then redistributed to the parallel processor responsible for clustering the respective subspace.

In *HPDBSCAN* the indexing is performed by employing a hashmap with offset and pointers into the data memory. For this, all parallel processors read an arbitrary, non-overlapping, equally-sized chunk of the complete dataset first. Then each data item of a chunk is uniquely associated with the cell of the overlayed grid that it spatially occupies, and vice versa—every grid cell contains all the data items that its bounding box covers. This in turn enables us to order all of the local data items with respect to their grid cell so that they are consecutively placed in memory. Finally, an indexing hashmap can be constructed with the grid cells being the key, and the tuple of pointer into the memory and number of items in this cell the value. An indexing approach like this has an additional memory overhead of $O(log(n))$ similar to other approaches like R- or k-d-trees. Figure 3 shows the indexing approach examplified by the dataset, introduced in Section 3, for the data chunk of a processing unit called `processor 1`.

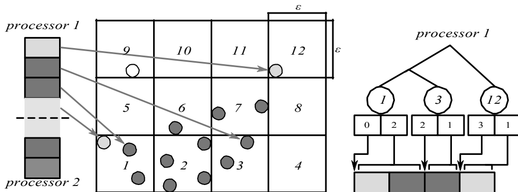


Figure 3: Sorted data chunks locally indexed by each processor using hashmaps pointing into the memory

Using that index the data redistribution can be performed in a straightforward fashion. The local data points of a parallel processor that do not lie within its assigned subspace are simply transferred to the respective parallel processor "owning" them. Afterwards all parallel processors have to rebuild their local data indices in order to encompass the received data. An efficient way of doing this is to send the section of the data index structure along with the data to the recipients. Due to the fact that the received and the local data are pre-sorted, the sent data index section and its memory pointers can be used to quickly merge them using, e.g., the merge-step of mergesort. The downside of the data redistribution approach is that it requires an additional memory overhead of $O(\frac{n}{p})$, per parallel processor, with p being the number of parallel processors, to be able to restore the initial data arrangement after the clustering. However, since the additional overhead has linear complexity, it is maintainable even for large scale problems.

Using the described index structure, cell-neighborhood queries execute in amortized computation time of $O(1)$. The cell-neighborhood N_{cell} thereby consists of all cells that are directly bordering the searched cell, its diagonals, as well as the cell itself with respect to all dimensions. For the cell labeled 6 in Figure 3 the cell-neighborhood is the set $\{1, 2, 3, 5, 6, 7, 9, 10, 11\}$. A formal definition follows.

Definition 9. Cell neighborhood —The cell neighborhood $N_{Cell}(c)$ of a given cell c denotes all cells d from the space of all available grid cells C that have a Chebychev distance $dist_{Chebychev}$ [6] of zero or one to c, i.e., $N_{Cell}(c) = \{d | d \in C \wedge dist_{Chebychev}(c, d) \leq 1\}$.

The actual epsilon neighborhood is then constructed from all points within the direct cell-neighborhood, filtered using the distance function *dist*. Šidlauskas et al. [26] show that a spatial grid index like this, is superior to R-trees and k-d-trees on index creation and queries, in terms of computation time, under the assumption that the cell granularity is optimal with respect to future neighborhood searches. Due to the fact that *DBSCAN*'s search radius is constant, the cells can trivially be determined to be hypercubes with the side length of ε. From a technical perspective it has the additional advantage that each of the d parts of the entire cell-neighborhood vector are consecutive in memory. This in turn enables data pre-fetching and the reuse of cell neighborhoods, thus reducing the number of cache misses.

In order to be able to answer all range queries within its assigned subspace, a parallel processor needs an additional one-cell-thick-layer of redundant data items, surrounding the grid splits that allow them to compute the cell neighborhood even at the edges of said splits. In parallel codes this is commonly referred to as *halos* or *ghost cells*. An efficient way of providing these halo cells is to transfer them along with the actual data during the data redistribution phase. This way the parallel processor will also index them along with the other data. Halo cells do not change the actual split boundaries in which a parallel processor operates and can be removed after the local clustering.

4.2 Cost heuristic

In the previous section we introduced the notion of subdividing the data space in order to efficiently parallelize *HPDBSCAN* and its spatial indexing especially also in distributed computing environments. However, we have not introduced a way to determine the boundaries of these splits. One of the most naïve approaches is to subdivide the space in equally-sized chunks in all dimensions, so that the resulting number of chunks equals to the number of available cores. While the latter part of the assumption is sensible as it minimizes the communication overhead, the former is not. Consider a spatially skewed dataset like shown in figure 4. The sketched dotted boundary, chunking the data space equally for two parallel processors, results in a highly unbalanced computational load, where one core needs to cluster almost all the data points and the other idles most of the time. Due to the fact that computing the *dist* function, while filtering the cell neighborhood, is for many distance functions the most processing intensive part of *DBSCAN*, this distribution pattern is particularly undesirable. It should also be clear that this is not only an issue of the presented example, but other spatially skewed datasets and larger processing core counts as well.

Therefore, we employ a cost heuristic to determine a more balanced data space subdivision. For this, we exploit the computation complexity properties of the cell neighborhood

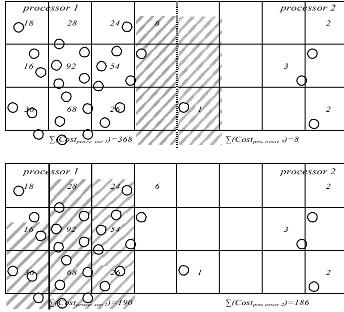


Figure 4: Impact of naive and heuristic-based hypergrid decompositions on compute load balancing. Halo cells are marked with a hatched pattern.

query. For each data item we have to perform n computations of the distance function *dist*, where n is the number of data items in the cell neighborhood. Since we have to do that for all m data items within a cell, the total number of comparisons for one cell is $n * m$. The sum of all comparisons, i.e. the cost scores, for all cells gives us the total "difficulty" of the whole clustering problem, at least in terms of the *dist* function evaluations. Then, we can assign to each parallel processor a consecutive chunk of cells, the cost of which is about a p-th part of the total score with p being the number of available parallel processing cores. The formal definitions are as follows.

Definition 10. Cell cost —The cell cost $Cost_{Cell}(c)$ of a cell c is the product of the number of items in it multiplied with the number of data points in the cell neighborhood – $Cost_{Cell}(c) = |c| * |N_{Cell}(c)|$.

Definition 11. Total cost —The total cost $Cost_{Total}$ is equal to the sum – $\sum_{c \in Cells} Cost_{Cell}(c)$ – of all individual cells.

Since the data items are already pre-sorted due to the spatial preprocessing step, the hypergrid subdivision can be performed by iteratively accumulating cell cost scores until the per-core threshold is reached or exceeded. Moreover, the the cell itself can be subdivided to gain more fine-grained control. For this, the cost score of the cell is not added entirely but in n-steps for each data item in the cell, where n is the number of items in the cell neighborhood. Figure 4 shows an example of a dataset, its overlayed hypergrid, the annotated cell cost values and the resulting subdivision. These subdivision can easily be computed in parallel by computing the cell score locally, reducing them to a global histogram and finally determining the boundaries according to the explained accumulative algorithm.

4.3 Local *DBSCAN*

Having redistributed the chunks among the compute nodes in a balanced fashion, the local *DBSCAN* execution follows. To break the need for sequential computation, implied by the recursive cluster expansion, this stage is converted to a parallelizable version with a single loop iterating over all data points. This enables the further, fine-grained parallelization of the algorithms using shared-memory parallelization approaches such as threads for example. The performance-focused algorithm redesign is twofold at this stage. Besides the parallelization of the iterations, the amount of computation per iteration is also minimized. Due to the cell-wise sorting and indexing of data points within the local data chunk, all points occupying one cell are stored consecutively in memory. This ensures that each cell-neighborhood must be computed at most once per thread, as each of them can be cached until all queries from within the same cell are visited. Listing 2 presents the pseudocode of the converted, iterative local *DBSCAN*.

```
1  def localDBSCAN(X, eps, minPts):
2    rules = Rules()
3    @parallel
4    for p in X:
5      Cp, Np = query(p, X, eps)
6      if length(Np) >= minPts:
7        markAsCore(p)
8        add(Cp, p)
9        for q in Np:
10         Cq = getCluster(q)
11         if isCore(q):
12           markAsSame(rules, Cp, Cq)
13         add(Cp, q)
14     elif notVisited(p):
15       markAsNoise(p)
16   return rules
```

Listing 2: Local *DBSCAN* pseudocode

For each of the points the epsilon neighborhood query is performed independently, i.e., not as an recursive expansion. When a query for a point p returns more than $minPoints$ data points, from which none is yet labeled, p is marked as a core of a cluster. The newly created cluster is then labeled using p's data point index, which is globally unique for the entire sorted dataset. If the epsilon neighborhood, numbering at least $minPoints$, contains a point q that is already assigned to a cluster, the point p is added to that cluster and inherits the cluster label from q. In case of multiple cluster labels present in the neighborhood, the core p inherits any one of the cluster labels and notes information indicating that each of the encountered subclusters actually are one, as they are inherently density connected. That information is vital to formulate merger rules for the subsequent merging of local cluster fragments and unification of cluster labels in the global scope (see section 4.4).

In all of the above cases, the remainder of non-core points in the epsilon neighborhood, which may also include halo area points, is added to the cluster of p. If p has less than $minPoints$ data points in its neighborhood, it is marked as visited and labeled as noise. The below proof shows that replacing the iterative cluster relabeling is equivalent to the original recursive expansion.

Theorem 1. Given points $p \in C_p$ and $q \in C_q$: $(Core(p) \vee Core(q)) \wedge DDR(p, q) \implies \exists C : C_p \cup C_q \subseteq C \wedge p, q \in C$

Proof. If neither p nor q is core, or they are mutually not *DDR*, the assumption is false and the implication trivially true. If p, q or both are cores, and they are *DDR* then by definition, they are also *DR* and therefore *DC*, with the linking point r being either p or q. Given the density connection *DC* between p and q, they belong to the same cluster C. By extension, any point belonging to C_p or C_q also belongs to C. □

The result of local *DBSCAN* is a list of subclusters along with the points and cores they contain, a list of noise points, and a set of rules describing which cluster labels are equivalent. This information is necessary and sufficient for the next step of merging the overlapping clusters with contradictory labels within the nodes' halos.

4.4 Rule-based cluster merging

The relabeling rules created by distinctive nodes are insufficient for merging cluster fragments from separate dataset chunks. The label-mapping rules across different nodes are created based on the labels of halo points. Upon the completion of the local *DBSCAN*, each halo zone is passed to the node that owns the actual neighboring data chunk. Subsequently, the comparison of local and halo point labels follows, resulting analogously in a set of relabeling rules for neighboring chunks, which may create transitive cluster label chains. These rules are then serialized and broadcast to all other nodes. Only then is the minimization of all local and inter-chunk label-mapping rules possible, and all transitive labels can be removed. Thus each compute node is equipped with a list of direct mappings from each existing subcluster label to a single global cluster label.

Each compute node then proceeds to relabel the owned clusters using the merger rules. At that stage each data point, now having assigned a cluster label, is sent back to the compute node that originally loaded it from the dataset. Recreation of the order of all data points is enabled by the initial ordering information created during the data redistribution phase. The distributed *HPDBSCAN* execution is complete and the result is a list of cluster ids or noise markers per data item.

5. IMPLEMENTATION

In this section we present our prototypical realization of *HPDBSCAN* and specifics of distinct technical details. The C++ source code can be obtained freely from our source code repository [23]. It depends on the parallel programing APIs Open Multiprocessing (OpenMP) [8] in version 4.0+ and Message Passing Interface (MPI) [15] in version 1.1+. Additionally, the command-line version requires the I/O library Hierarchical Data Format 5 (HDF5) [18] in order to pass the data and store computational results.

5.1 Data distribution and gathering

As explained in section 4.1, the data items of the datasets are redistributed in the preprocessing step, in order to achieve data locality. Implementing this behavior in shared-memory architectures is trivially not required, due to the fact that all processors can access the same memory. For distributed environments, however, this step is needed and can be quite challenging to realize—especially in a scalable fashion.

Since *HPDBSCAN* sorts the data points during the indexing phase and lays them out consecutively in memory, we are able to exploit collective communication operations of the MPI. We first send the local histograms of data points from each compute node to the one that owns the respective bounds during the local *DBSCAN* execution. This can be implemented either by an `MPI_Reduce` or, alternatively, by an `MPI_Alltoall` and a subsequent summation of the array. After that, each of the compute nodes allocates local memory, and the actual data points are exchanged using an `MPI_Alltoallv` call. Using the received histograms, the compute nodes are also able to memorize the initial ordering of the data points, in a flat array, for example.

Vice versa, the gather step can be implemented analogously. Instead of sorting the local data items by their assigned grid cell, they are now re-organized by their initial global position in the dataset. After that, they can be exchanged again, using the MPI collectives, and stored. Note that in this step the computed cluster labels are transferred along with the data points in order to avoid multiple communication.

5.2 Lock-free cluster labeling

To ensure that the cluster labels are unique within a chunk as well as globally, each cluster label c is determined by the lowest index of a core point inside a cluster—$c = \min_{p \in C | Core(p)} index(p)$. The $index(p)$ function returns the position of a data point p, within the globally sorted dataset, redistributed to the compute nodes. Additionally to ensuring global uniqueness, this mechanism also maximizes the size of consistently labeled cluster fragments within the same compute node, as each consecutive iteration over the points increments the current point's index. Whenever a core is found in the epsilon neighborhood, the current point inherits its cluster label, even if it is a core itself.

A data race may occur, when the current epsilon neighborhoods of multiple parallel threads overlap. In that case each thread may attempt to assign a label to a point within their neighborhood intersection. The naïve approach of locking the data structures storing the cluster label and core information is not scalable.

The better alternative of using atomic operations, here atomic *min*, requires encoding the values to operate on, with a single native data type. For this, we use signed long integer type values, and compress all flags and labels described by *DBSCAN*'s original definition, i.e., "visited", "core" and "noise" flags, and a "cluster label", to that data type. As the iterations are performed for each data point exactly once, the "visited" flag, is made redundant and abandoned. The cluster label value is stored using the absolute value of the lowest core point index it contains. The sign bit is used to encode the "core" flag, such that each core of cluster c is marked by value -c, and each non-core point—by value c. As cluster labels are created using point indexes, their value never exceeds $|X|$. The noise label can then be encoded using any value from outside the range $[-|X|, |X|]$. For this, we have selected the upper bound of the value range—the maximal positive signed long integer. As long as $range(signed_long_int) >= |X| + 1$, signed long integers are sufficient to encode the cluster labels as well as the core and noise flags. In that way, minimizing the cluster label is possible via simple atomic min implementation to set the cluster label and core flag at once. Some processor architectures, e.g., Intel x86, do not provide an atomic *min* instruction. Instead, a spinlock realization using basic atomic read and compare-and-swap instruction, as shown in Listing 3, is used.

```
1 def atomicMin(address, val):
2   prev = atomicRead(address)
3   while(prev > val):
4     swapped = CAS(address, prev, val)
5     if swapped: break
6     prev = atomicRead(address)
```

Listing 3: Spinlock atomic *min*

5.3 Parallelization of the local *DBSCAN* loop

The iterative conversion of *DBSCAN* allows us to divide the computation of the loop iterations among all threads of a compute node. Because the density of data points within a chunk can be highly skewed, a naive chunking approach is suboptimal (see Section 4.2), and can lead to a highly unbalanced work load. To mitigate this, a work stealing approach is advisable. Our *HPDBSCAN* implements threading using OpenMP's `parallel for` pragma. The closest representative of work stealing in OpenMP is the `schedule(dynamic)` clause, added to the `parallel for` pragma. Optimal performance is achieved, when the dynamically pulled workload is small enough—so that the workload imbalances are split and fairly divided, and at the same time large enough—so that not too many atomic *min* operations (whether supported by hardware or not) are performed simultaneously on the same memory location. This number is highly dependent on environment details, such as the clustered problem and the execution hardware. Through empirical tests, however, we determined a reasonable dynamic chunk size of 40.

6. EXPERIMENTAL EVALUATION

In this section we will describe the methodology and findings of the experiments conducted to evaluate the parallel *DBSCAN* approach described above. The main focus of the investigation is the performance evaluation of the implementation with respect to computation time, memory consumption and the parallel programming metrics: speed- and scale-up [12].

6.1 Hardware setup

To verify the computation time and speed up of our implementation, we have performed tests on the Juelich Dedicated Graphic Environment (JUDGE) [10]. It consists of 206 IBM System x iDataPlex dx360 M3 compute nodes, where each node has 12 compute cores combined through two Intel Xeon X5650 (Westmere) hex-core processors clocked at 2.66 GHz. A compute node has 96 GB of DDR-2 main memory. JUDGE is connected to a parallel, network-attached GPFS-storage system, called Juelich Storage Cluster (JUST) [11]. Even though the system has a total core count of 2.472, we were only able to acquire a maximum of 64 nodes (768 cores) for our benchmark, as JUDGE is used as a production cluster for other scientific applications. Our hardware allocation, though, was solely dedicated for us, which ensured that no other computations interfered with our tests. The plugged-in Westmere processors allow to use 24 virtual processors, when hyperthreading is enabled. For the test runs, however, we disabled this feature as it can falsify or destabilize measurement correctness, as Leng et al. [22] have shown. In a multithreading scenario we facilitate for this reason a maximum of 12 threads per node.

6.2 Software setup

The operation system running on JUDGE is a SUSE Linux SLES 11 with the kernel version 2.6.32.59-0.7. All applications in the test have been compiled with gcc 4.9.2 using the optimization level O3. The MPI distribution on JUDGE is MPICH2 in version 1.2.1p1. For the compilation of *HPDBSCAN*, a working HDF5 development library including headers and C++ bindings is required. For our benchmarks we used the HDF group's reference implementation, version 1.8.14, pre-installed on JUDGE. Later in this section we present a comparison of *HPDBSCAN* with *PDSDBSCAN-D* created by Patwari et al [25]. The latter needs the parallel netCDF I/O library. We have obtained and compiled pnetCDF from the project's web page at Northwestern University with version 1.5.0 [24].

Dataset	Points	Dims.	Size (MB)	ε	$minPts$
Tweets [*t*]	16,602,137	2	253.34	0.01	40
Twitter small [*ts*]	3,704,351	2	56.52	0.01	40
Bremen [*b*]	81,398,810	3	1863.68	100	10000
Bremen small [*bs*]	2,543,712	3	48.51	100	312

Table 1: *HPDBSCAN* benchmark datasets properties

6.3 Datasets

Despite *DBSCAN*'s popularity its parallelization attempts were mainly evaluated using synthetic datasets. To their advantage, they can provide an arbitrarily large number of data points and dimensionality. The downside, however, is that they are not representative for actual real world applications. They might have inherent regular patterns from, e.g., pseudo random number generators that will silently bias the implementation's performance. For this reason, we decided to resort to actual real-world data and its potential skew. An overview of the chosen examples is depicted in Table 1. We acknowledge that an evaluation of higher-dimensional datasets is of great interest for some clustering application, such as, for instance, genomics in bio-informatics, but could not be obtained at the time of writing.

6.3.1 Geo-tagged collection of tweets

This set was collected and made available to us by Junjun Yin form the National Center for Supercomputing Application (NCSA). The dataset was obtained using the free twitter streaming API and contains exactly one percent of all geo-tagged tweets from the United Kingdom in June 2014. It was initially created to investigate the possibility of mining people's trajectories and to identify hotspots and points of interest (clusters of people) through monitoring tweet density. The full collection spans roughly 16.6 million tweets. A smaller subset of this was generated by filtering the entire set for the first week of June only. Both datasets are available at the scientific storage and sharing platform B2SHARE [14].

6.3.2 Point cloud of Bremen's old town

This data was collected and made available by Dorit Borrmann and Andreas Nüchter from the Institute of Computer Science at the Jacobs University Bremen, Germany. It is a 3D-point cloud of the old town of Bremen. A point cloud is a set of points and its representing coordinate system that often model the surface of objects. This particular point cloud of Bremen was recorded using a laser scanner system mounted onto an autonomous robotic vehicle. It has stopped at 11 different locations, performing each time a full 360° scan of the surrounding area. Given the GPS triangulated position and perspective of the camera, the sub-point clouds were combined to one monolith. The raw data is available from Borrmann and Nüchter's webpage [20]. An already combined version in HDF5 format, created by us, can be obtained from B2SHARE [14]. *DBSCAN* can be applied here in order to clean the dataset from noise or outliers, such as falsy scans or unwanted reflections of moving

objects. Moreover, *DBSCAN* can also be used to find distinct objects, represented as clusters, in the point cloud like houses, roads or people. The whole point cloud contains roughly 81.3 million data points. A smaller variant was generated by randomly sampling $\frac{1}{32}$ of the points that is also available on B2SHARE [14].

6.4 Speed up evaluation of *HPDBSCAN*

We benchmark our *HPDBSCAN* application's speed up using both, the full Twitter (t) and the full Bremen (b) dataset. Our principal methodological approach is thereby as follows. Each benchmark is ran five times, measuring the application's walltime at the beginning and end of the `main()` function of the process with the MPI rank 0 and the OpenMP thread number 0. After these five runs we double the number of nodes and cores, starting from one node and 12 cores, up to the maximum of 768 cores. In addition to that we have run a base measurement with exactly one core on one node. For each "five-pack" benchmark run we report the minimum, maximum, mean μ, standard deviation σ and coefficient of variation (CV), defined as $v = \frac{\mu}{\sigma}$ [1]. The speed up coefficient is calculated in comparison to the single core run, based on the mean values of the measurements for each processor count configuration. Both datasets are processed using the OpenMP/MPI hybrid features of our application. That means that we spawn an MPI process for each node available and parallelize locally on the nodes using OpenMP. For the Bremen point cloud we have additionally parallelized the computation with MPI alone, i.e., we use one MPI process per core, enabling direct comparison of the hybrid and fully distributed versions.

Nodes		1	1	2	...	32	64
Cores		1	12	24		384	768
OpenMP+MPI hybrid b							
Mean	μ	**79372.29**	**8037.71**	**4271.64**		**327.07**	**172.53**
time (s) StDev	σ	17.6011	71.2829	16.2092		2.5971	1.3801
CV	υ	0.00022	0.00886	0.00379		0.0079	0.0079
Min		79342.08	7937.48	4253.45		322.71	170.77
Max		79385.57	8129.85	4293.86		329.65	174.47
Speed-Up		**1**	**9.9**	**18.6**		**242.7**	**460.0**
MPI b							
Mean	μ	**79372.29**	**8028.67**	**4403.96**		**515.21**	**354.99**
time (s) SrDev	σ	17.6012	9.5769	7.1526		94.7806	42.0006
CV	υ	0.00022	0.00119	0.00162		0.18396	0.11832
Min		79342.08	8019.10	4395.78		471.10	302.27
Max		79385.57	8040.83	4415.45		684.74	420.01
Speed-Up		**1**	**9.9**	**18.0**		**154.1**	**232.7**
OpenMP+MPI hybrid t							
Mean	μ	**2079.26**	**212.77**	**115.66**		**10.04**	**7.88**
time (s) StDev	σ	1.06455	0.56826	0.35893		0.42128	1.03302
CV	υ	0.00051	0.00267	0.00310		0.04194	0.13106
Min		2078.16	212.05	115.34		9.76	7.14
Max		2080.47	213.43	116.17		10.78	9.70
Speed-Up		**1**	**9.8**	**18.0**		**207.0**	**263.8**

Table 2: Measured and calculated values of the *HPDBSCAN* speed-up evaluation

The results in Table 2 and Figure 5 show that we are able to gain substantial speed up for both data sets. It peaks for Bremen at 460.0 using 768 cores, and in the Twitter analysis case at slightly more than half of this value at 263.8. For the MPI-only clustering of the Bremen dataset the speed up value falls short of the hybrid implementation, being only roughly half of it with 232.7 using 768 cores. There are two noteworthy facts that can be observed in the measurement data. The first and obvious one is that the hybrid implementation outperforms the fully distributed MPI runs by a factor of two. The access to a shared cell index and the reduced number of nodes to communicate it to, significantly reduces communication overhead and enables faster processing time. Secondly, one can observe a steady decrease in the efficiency of additional cores used for the clustering. This seems to be especially true for the tweet collections compares to the Bremen dataset. This observation can be explained best through Amdahl's law [2]. In the benchmark we use a constant problem size, disallowing infinite speed up performance gains. Instead, we approach the asymptote of the single threaded program parts. Due to the fact that tweet collection is smaller in size, we approach this boundary earlier than with the Bremen data for instance. Additional network communication overhead with larger processor counts, `atomicMin()` clashes as well as load imbalances are good examples of simultaneously growing serial program parts. Moreover, the growing CV value is a good indicator for the increasing influence of external factors onto the measurements, like varying operating system scheduling.

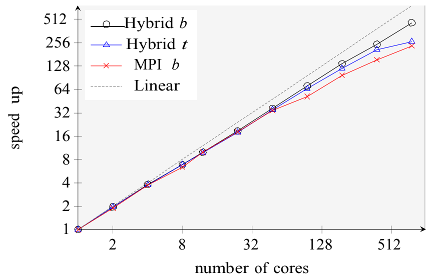


Figure 5: Speed up curves of the *HPDBSCAN* application analyzing the Bremen and Twitter datasets

6.5 Scale up evaluation of *HPDBSCAN*

In this section we investigate *HPDBSCAN*'s scalability properties. Our principal measuring methodology remains unchanged. Instead of the speed up coefficient, we report the efficiency value $e_p = \frac{t_1}{t_p}$ for each benchmark run, which is the fraction of the execution time with a single core and the execution time with p processing cores. Perfect scalability is achieved, when the efficiency equals one or is almost one. Yet, it requires doubling the dataset size, whenever we double the processor count.

As a base for this we use the small Bremen dataset bs. In particular, for each run, we copy the entire data p times, where p is equal to the number of used processors. Then, each copy is shifted along the first axis of the dataset by the copy's index, times the axis range, and concatenated with the others. This way, we get multiple (p) Bremen old towns next to one another. We chose this approach to get a better grasp of the overhead of our implementation by presenting the same problem to each available MPI process. In contrast to that, a random sampling of the whole Bremen dataset, for instance, would have altered the problem difficulty.

The results of our test can be seen in Table 3 and Figure 6. In all of the three scenarios a near constant efficiency

		Cores	1	2	4	8	16	32
				OpenMP				
time (s)	**Mean**	μ	**99.5044**	**102.644**	**107.462**	**121.35**	-	-
	StDev	σ	0.0991	0.0439	0.1515	5.6788	-	-
	CV	υ	0.00099	0.00042	0.00141	0.04679	-	-
	Min		99.41	102.58	107.2	117.81	-	-
	Max		99.66	102.69	107.59	131.45	-	-
				MPI				
time (s)	**Mean**	μ	**99.50**	**101.86**	**103.74**	**105.48**	**107.12**	**109.19**
	StDev	σ	0.0991	0.0884	0.1131	0.1881	0.5399	0.6653
	CV	υ	0.00099	0.00086	0.00109	0.00178	0.00504	0.00609
	Min		99.41	101.79	103.65	105.3	106.56	108.52
	Max		99.66	102	103.91	105.76	107.77	110.15

		Cores	12	24	48	96	192	384
				OpenMP/MPI hybrid				
time (s)	**Mean**	μ	**10.46**	**13.3**	**14.02**	**14.786**	**16.31**	**19.76**
	StDev	σ	0.0974	0.1425	0.2420	0.2548	0.58307	3.15081
	CV	υ	0.00931	0.01071	0.24207	0.25481	0.03578	0.15943
	Min		10.38	13.12	13.76	14.57	15.67	17.909
	Max		10.63	13.45	14.28	15.16	17.12	25.26

Table 3: Measured and calculated values of the *HPDBSCAN* scale-up evaluation of the Bremen data

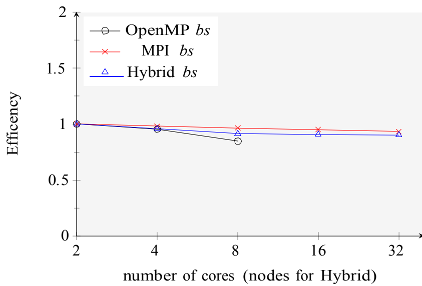


Figure 6: Scale up curves of *HPDBSCAN* analyzing the Bremen and Twitter datasets

value can be achieved, indicating good scalability. While the MPI-only and OpenMP/MPI hybrid benchmark runs only have a slightly increasing execution time curve, we can observe a clear peek for the OpenMP benchmark with four and more cores. Through a separate test, we can attribute this increase to more contentions in the spinlock of our `atomicMin()` implementation, introduced in Section 5.

6.6 Comparison of *HPDBSCAN* and *PDSDBSCAN*

As discussed in related work there are a number of other parallel versions of *DBSCAN*. Most of them report different value permutations for the computation time, memory consumption, speed up and scalability of their implementations. Almost all of them do not to provide either their used benchmark datasets or the source code of their implementations. This in turn, disallows us to verify their results or to perform a direct comparison of our approaches. To the best of our knowledge the only exception are Patwary et al. [25]. Their datasets can also not be recreated, but they made the C++ implementation of their parallel disjoint-set data structure *DBSCAN* —*PDSDBSCAN* —open-source. This allows us to compare our approach with theirs at least using our benchmark datasets. Patwary et al. offer two versions of *PDSDBSCAN*. One targets shared-memory architectures only and is based on OpenMP. The other can also be used in distributed environments and is implemented using MPI. In order to distinguish between these two, the suffixes -S or -D are added respectively.

In order to compare both parallel *DBSCAN* approaches, we have performed another speed up benchmark according to the introduced methodology on the small Twitter dataset. Due to their technical similarities our two "contestants" are *HPDBSCAN* using MPI processes only and *PDSDBSCAN-D*. Thereby, we scale the process count from one to a maximum of 32, each being executed on a separate compute node. Even though we have executed five runs for each level of used processors, we report here only the mean value for execution time, memory consumption and speed up, because of space considerations.

Nodes	1	2	4	8	16	32
			HPDBSCAN MPI			
time (s)	114.39	58.99	30.14	15.71	8.37	6.07
Speed-Up	1.00	1.94	3.80	7.28	13.67	18.85
Memory (MB)	251064	345276	433340	678248	1101000	2111000
			PDSDBSCAN-D			
time (s)	288.35	162.47	105.94	89.87	85.37	88.42
Speed-Up	1	1.77	2.72	3.21	3.38	3.36
Memory (MB)	500512	725104	1370000	4954000	19724000	59685000

Table 4: Comparison of *HPDBSCAN* and $PDSDBSCAN-D$ using the Twitter dataset

Table 4 and Figure 7 present the obtained results. *HPDBSCAN* shows a constant, near linear speed-up curve, whereas *PDSDBSCAN-D* starts similarly, but soon flattens, stabilizing at a speed-up of around 3.5. The curve for the memory consumption is inverse. *HPDBSCAN* shows a linear increase again, seemingly being dependent on the number of used processing cores, which can be explained by the larger number of replicated points in the halo areas. *PDSDBSCAN-D*, however, presents an exponential memory consumption. An investigation of the source code reveals that each MPI process always loads the entire datafile into main memory, effectively limiting its capabilities to scale with larger datasets. This is also the reason why we have used the small Twitter dataset *ts* for this experiment, as larger datasets have caused out-of-memory exceptions. As a consequence, we have not been able to reproduce the performance capabilities of *PDSDBSCAN-D*.

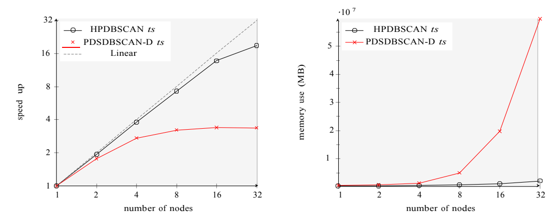


Figure 7: Speed up and memory usage of *HPDBSCAN* compared to $PDSDBSCAN-D$

7. CONCLUSION

In this paper, we have presented *HPDBSCAN*—a scalable version of the density-based clustering algorithm *DB-*

SCAN. We have overcome the algorithm's inherent sequential control flow dependencies through a divide-and-conquer approach, using techniques from cell-based clustering algorithms. Specifically, we employ a regular hypergrid as the spatial index in order to minimize the neighborhood-search spaces and to partition the entire cluster analysis into local subtasks, without requiring further communication. Using a rule-based merging scheme, we combine the found local cluster-labels into a global view. In addition to that, we also proposes a cost heuristic that allows to balance the computation workload, facilitated by the previously mentioned cells, divided among the compute nodes according to their computation complexity. We have implemented *HPDBSCAN* as an open-source OpenMP/MPI hybrid application in C++, which can be deployed in shared-memory as well as distributed-memory computing environments. Our experimental evaluation of the application has proven the algorithm's scalability in terms of memory consumption and computation time, outperforming *PDSDBSCAN*, the first parallel HPC implementation. The presented cell-based spatial index can easily be transferred to other clustering and neighborhood-search problems with constant search range. In future work we plan to demonstrate this on the basis of parallelizing other clustering algorithms, such as *OPTICS* and *SUBCLU*.

8. REFERENCES

[1] H. Abdi. Coefficient of variation. *Encyclopedia of research design*, pages 169–171, 2010.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference 1967*, pages 483–485. ACM, 1967.

[3] D. Arlia and M. Coppola. Experiments in parallel clustering with dbscan. In *Euro-Par 2001 Parallel Processing*, pages 326–331. Springer, 2001.

[4] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[5] S. Brecheisen, H.-P. Kriegel, and M. Pfeifle. Parallel density-based clustering of complex objects. In *Advances in Knowledge Discovery and Data Mining*, pages 179–188. Springer, 2006.

[6] C. Cantrell. *Modern mathematical methods for physicists and engineers*. CUP, 2000.

[7] M. Chen, X. Gao, and H. Li. Parallel DBSCAN with Priority R-Tree. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference*, pages 508–511. IEEE, 2010.

[8] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[10] Forschungszentrum Jülich GmbH. Juelich Dedicated GPU Environment. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUDGE/JUDGE_node.html`.

[11] Forschungszentrum Jülich GmbH. Juelich Storage Cluster. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/OnlineStorage/JUST/JUST_node.html`.

[12] I. Foster. *Designing and building parallel programs*. Addison Wesley Publishing Company, 1995.

[13] Y. X. Fu, W. Z. Zhao, and H. F. Ma. Research on parallel dbscan algorithm design based on mapreduce. *Advanced Materials Research*, 301:1133–1138, 2011.

[14] Götz, Markus and Bodenstein, Christian. HPDBSCAN Benchmark test files. `http://hdl.handle.net/11304/6eacaa76-c275-11e4-ac7e-860aa0063d1f`.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[16] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, et al. The universe at extreme scale: multi-petaflop sky simulation on the bg/q. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 4. IEEE Computer Society Press, 2012.

[17] J. Han, M. Kamber, and J. Pei. *Data Mining: concepts and techniques - 3rd ed.* Morgan Kaufmann, 2011.

[18] HDF Group. Hierachical Data Format 5. `http://www.hdfgroup.org/HDF5`.

[19] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. MR-DBSCAN: an efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems, 2011 IEEE 17th International Conference*, pages 473–480, 2011.

[20] Jacobs University Bremen. 3D Scan Repository. `http://kos.informatik.uni-osnabrueck.de/3Dscans/`.

[21] A. Knöpfel, B. Gröne, and P. Tabeling. *Fundamental modeling concepts*, volume 154. Wiley, UK, 2005.

[22] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.

[23] Markus Götz. HPDBSCAN implementation. `https://bitbucket.org/markus.goetz/hpdbscan`.

[24] Northwestern University. Parallel netCDF. `http://cucis.ece.northwestern.edu/projects/PnetCDF/`.

[25] M. M. A. Patwary, D. Palsetia, A. Agrawal, et al. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), International Conference for*, pages 1–11. IEEE, 2012.

[26] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 236–245. ACM, 2009.

[27] X. XU, J. JAGER, and H.-P. KRIEGEL. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery*, 3:263–290, 1999.

[28] A. Zhou, S. Zhou, J. Cao, Y. Fan, and Y. Hu. Approaches for scaling dbscan algorithm to large spatial databases. *Journal of computer science and technology*, 15(6):509–526, 2000.

Appendix C

Paper III

2016 15th IEEE International Conference on Machine Learning and Applications

Automatic Object Detection using DBSCAN for Counting Intoxicated Flies in the FLORIDA Assay

Christian Bodenstein*‡, Markus Götz*‡, Annika Jansen†, Henrike Scholz†, Morris Riedel*‡
* Juelich Supercomputing Center, Leo-Brandt-Straße, 52425 Juelich, Germany
{c.bodenstein, m.goetz, m.riedel}@fz-juelich.de
† University of Cologne, Department of Biology, Zuelpicher Straße 47b, 50674 Cologne, Germany
{annika.jansen, henrike.scholz}@uni-koeln.de
‡ University of Iceland, Sæmundargötu 2, 101 Reykjavik, Iceland

***Abstract*—In this paper, we propose an instrumentation and computer vision pipeline that allows automatic object detection on images taken from multiple experimental set ups. We demonstrate the approach by autonomously counting intoxicated flies in the FLORIDA assay. The assay measures the effect of ethanol exposure onto the ability of a vinegar fly *Drosophila melanogaster* to right itself. The analysis consists of a three-step approach. First, obtaining an image of a large set of individual experiments, second, identify areas containing a single experiment, and third, discover the searched objects within the experiment. For the analysis we facilitate well-known computer vision and machine learning algorithms—namely color segmentation, threshold imaging and DBSCAN. The automation of the experiment enables an unprecedented reproducibility and consistency, while significantly decreasing the manual labor.**

***Index Terms*—Image Analysis, Computer Vision, Machine Learning, FLORIDA Assay, DBSCAN, Biology, Flies, Genetics**

I. Introduction

Object detection is one of the fundamental problems in computer vision. It is concerned with identifying objects in an image or video irrespective of variations to it, like for example different viewpoints, scaling, rotation, translation or partial obstruction. The literature proposes a multitude of approaches to tackle the problem, such as feature matching, template matching or machine learning. In this paper we propose a method that is based on the latter, namely, the unsupervised clustering algorithm DBSCAN and threshold imaging.

We demonstrate the approach by counting intoxicated vinegar flies *Drosophila melanogaster*. The data is generated with the FLORIDA assay, which measures the effect on ethanol onto the ability to right itself again after intoxication. This is used to identify genes and neuronal mechanisms underlying the intoxication effect. While we focus on the automation of the FLORIDA assay, the pipeline could be generalized to similar detection problems e.g. bacterial culture observation on well plates.

The remainder of this paper is structured as follows. Section II reviews related work, and Section III presents background information on the FLORIDA assay and the used algorithms. In sections IV and V the analysis process is presented in detail along with its implementation. An evaluation of the approach based on experimentation data is discussed in section VI and, finally, the paper is concluded in section VII followed bv an outlook on improvements in section VIII.

II. Related Work

Clustering is an established method for detecting and segmenting objects in images and videos. Some of the earlier attempts date back to the 1970's and 1980's. Coleman et al. [7] for instance propose an image segmentation method based on K-Means [13], one of the fundamental clustering algorithms, that is able to split a complete image into segments in the respective color space. Similar research has been conducted by Haralick et al. [14] in 1981 which is based upon hierarchical clustering and its different linkage types [13]. A good summary of these earlier attempts is given by Kettaf et. al [16].

In biology and bio-medicine K-Means, and its adaptive and fuzzy variants, are in widespread use, because of its robustness to low image resolution and noise in the images. Examples include tumor detection [18], volumetric reconstruction of the left ventricle chamber [6] or automatic identification of brain regions on MRT images [21]. More recently, density-based approaches have become popular. Unlike K-Means based approaches, they are able to directly detect non-circular objects in images, without having to piece them together from a set of sub-clusters, while at the same not requiring an exact number k of clusters to be identified. Celebi et al. [5] for instance apply DBSCAN [10] to segment an arbitrary amount of irregular skin lesions in dermatological imagery. For our problem at hand both of the above properties are desirable as we, first, do not know the number of individual objects, second, observe objects of arbitrary shapes. A more generalized method for density-based object segmentation in images is given by Ye et al [24] upon which we have based our approach.

III. Background

This section provides an overview of the experimental set up for data generation and used algorithms.

A. FLORIDA Assay

The "**F**ull **L**oss **O**f **R**ighting Reflex **I**n**D**uced by **A**lcohol" - FLORIDA assay is used to identify genes and neuronal networks underlying the ethanol induced loss of righting reflex, a measure for the degree of intoxication. The experimental

978-1-5090-6167-9/16 $31.00 © 2016 IEEE
DOI 10.1109/ICMLA.2016.49

animal is the common vinegar fly *Drosophila melanogaster*, that is a useful genetic tool to study the mechanistic basis of behaviors associated with alcoholism [22]. The vinegar flies not only share common genes but also behavioral similarities when intoxicated, and are therefore a useful genetic model system to understand ethanol induced behaviors in humans. Through alteration of genes in the specimen and subsequent observation of mutants, it is possible to analyze changes of behaviors that are associated with alcohol abuse disorders [22]. Concretely, the experiment is focused on the investigation of the flies' tolerance to the intoxication effect of ethanol. In line with that, a population of flies is exposed to vaporizing ethanol in experimentation vials. The individuals' intoxication level is recorded over time, by counting sedated flies. The sedation is defined as flies that fail to right themselves after a mechanic stimulation. This measure is also known as loss of righting reflex. The experimentation procedure is summarized as follows:

1) Multiple individuals, currently a group of 20, are placed into an experimentation vial. 2) The vial is closed by an ethanol soaked tissue which vaporizes over time. 3) In a specified interval—currently 60 seconds—the vials are shaken to startle the flies and identify the sedated flies that fail to right themselves. 4) The number of intoxicated objects needs to be determined in an image. 5) The experiments is terminated when all flies are sedated. 6) The data analysis has to be obtained for multiple vials containing groups of flies that are treated in parallel.

B. DBSCAN

Clustering algorithms in the field of machine learning are used to aggregate similar objects into common groups. *DBSCAN* is a particular, density-based clustering algorithm that was published 1996 by Ester et al. [10]. Its principal idea is to find dense areas and to expand these recursively in order to find clusters. A dense region is thereby formed by a point that has within a given search radius ϵ at least θ neighboring points, whereas θ is called density threshold, or in literature often referred as $minPoints$.

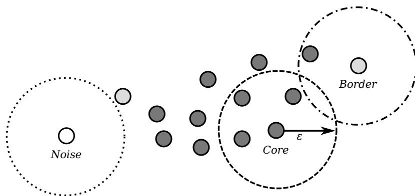


Fig. 1. $DBSCAN$ clustering with $minPoints$ $\theta = 4$ and search radius ε.

This dense area is also called the *core* of a cluster. For each of the found neighbor points the density criteria is reapplied and the cluster is consequently expanded. All points that do not form a cluster core and that are not "absorbed" through expansion are regarded as *noise*. A more formal definition can be found in [10]. Two of the major advantages of DBSCAN compared to other traditional clustering algorithms, like K-Means [13] for example, is that it can detect arbitrarily shaped clusters without having to know the number of clusters apriori.

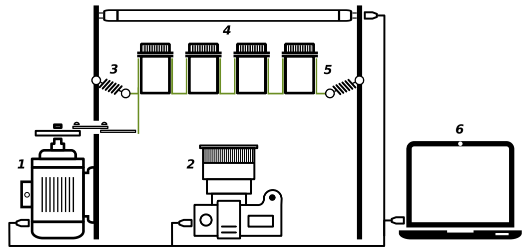


Fig. 2. The automated experimentation setup. It is consisting of (1) an electric engine, shaking the vials via a rotary crank, (2) a digital SLR, (3) a spring suspension allowing horizontal motion, (4) a top lightning plate, (5) the experimentation vials containing the fruit flies and (6) a notebook with the analysis software, connected via USB to other devices.

IV. Analysis Process

Counting the number of intoxicated flies is achieved in a three-step sequential analysis process. First, a high resolution image is retrieved from the FLORIDA laboratory setup's single-lens reflex (SLR) camera. Then, the vials are segmented to enable individual fly counting for each vial. Finally, after thresholding the image, the DBSCAN algorithm is used to cluster the darker image areas. The result is then used to count the number of flies in the image. Details of these steps are described in the sections below.

A. The FLORIDA laboratory setup

The FLORIDA laboratory setup, illustrated in Figure 2, has been built in order to enable the fully automatic counting of sedated flies and consists of several sub-parts. The main part is the holder plate for the experimentation vials, attached to the outer framework through metal springs which allow horizontal motion. The holder plate exhibits a grid of four by five vial slots, permitting a analysis of 20 vials in parallel with each vial equally holding 20 *drosophila melanogaster*.

The bottom of the holder plate is plastered with green foil, enabling accurate segmentation of the vial slots. A SLR camera is placed below the holder plate, with the lens faced to the bottom of the vials. Sedates flies gathering at the vials' bottoms can easily be captured by the camera, while flies in upper regions fade out due to the opaque vial material. To achieve a high contrast between flies and vial surface, a LED light plate is placed above the holder plate to screen the vials. As shown in Figure 2, an electronic engine is connected to the holder plate, to shake the vials before the photo is taken as to test the righting reflex. The SLR camera, the shaker engine and the light plate are connected via USB to a PC and can be controlled with our FLORIDA software. This includes capturing and streaming images from the camera directly to the program.

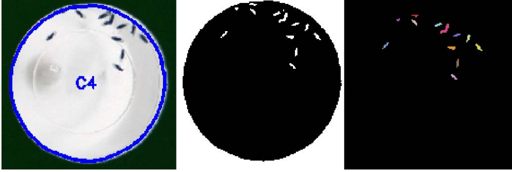


Fig. 3. The image processing steps from left to right—left: vial segmentatio;, center: threshold image; right: resulting clusters.

B. Vial Segmentation

Before the actual flies can be detected, the vials have to be found. Our first attempt was to use the Hough Transform [8] for finding circles. While this method was able to find the vials quickly, it does not provide the desired precision. Perspective distortions of the camera let the vials shapes appear more elliptical than circular. This is a problem that is difficult, while not impossible to manage with this algorithm.

To provide fast and accurate vial segmentation a green foil was glued on the underneath of the holder plate. This enables the usage of simple color segmentation techniques based on green screening (similar to what is done in movies). For this, the image has to be converted from RGB (Red Green Blue) to HSV (Hue Saturation Value) [11] color space, since it separates the color information from intensity and illumination. The hue value describes the color which should be in range $h = [40, 80]$ for green, the saturation, or color intensity, should be at least in range $s = [150, 255]$ and the value, or illumination is between $v = [10, 255]$, to avoid the segmentation of complete dark regions. After a mask is created that selects all pixels within these ranges, contours are detected using the Teh-Chin chain approximation algorithm [23]. All contours that contain an area of around $a = \pi\psi^2$, where ψ is the predefined vial radius, are considered as vials.

The resulting vials mask is post-processed with an erosion algorithm to smooth the edges. Furthermore, the contour points are moved few pixels towards the image center where outer points are stronger affected than inner points. This *rubber band effect* has the purpose to correct the different perspective view angles on the vials. The results are stored as polygons which can be used to create an image mask for the associated vial.

C. Threshold Clustering

After the vial segmentation step, each vial can now be analyzed separately. For this, the RGB image is converted first into a gray scale image and then the vial mask is used to whiten everything but the considered vial. Then, a threshold τ is defined. Given the image width X and image height Y, all pixels $p_{x,y}$ in the resulting image P are binarized using the following function:

$$p_{x,y} = \begin{cases} 255 & \text{if } p_{x,y} \leq \tau \\ 0 & \text{if } p_{x,y} > \tau \end{cases} \forall x \in [1, X] \forall y \in [1, Y] \quad (1)$$

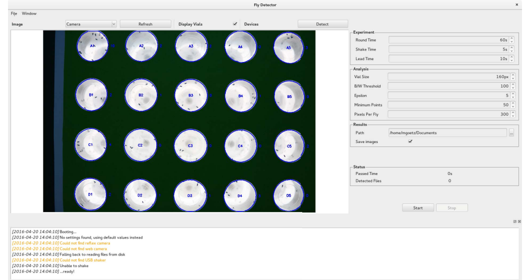


Fig. 4. The graphical user interface of the FLORIDA software with the to be analyzed image on the left and parameter settings on the right

This means, all pixels below the threshold are considered to belong to a fly and are mapped to white, while all other pixels are mapped to black. The coordinates of the white pixels W are then extracted:

$$W = \{(x, y) | \forall x \in [1, X] \forall y \in [1, Y] : p_{x,y} > 0\} \quad (2)$$

These are then fed to DBSCAN:

$$C, N = \text{DBSCAN}(\epsilon, \theta, W) \quad (3)$$

Whereby C are the resulting cluster labels and N the pixels considered to be noise.

D. Fly Counting

Finally, the found clusters C have to be counted. The naïve approach is to consider each cluster to represent a single fly. However, due to being close to one another, a single cluster may contain multiple flies, as seen in Figure 3, this will result in miscounts. Therefore, the fly count $\hat{f}$ is predicted as such:

$$\hat{f} = \sum_{i}^{|C|} \left\lceil \frac{|C_i|}{\rho} \right\rceil \quad (4)$$

With ρ being an additional pixels per fly parameter, defining the amount of cluster pixels ($|C_i|$) that will count as single fly. While more complex algorithms have been considered to overcome the problem, this solution is fast and yields sufficient performance. The threshold clustering and fly counting steps are repeated for each found vial in the image.

V. Implementation

The entire analysis process is implemented as part of a standalone GUI application that is operated by the laboratory assistants of the FLORIDA experiment. Figure 4 depicts an example of the interface, while the next section describe details.

A. Controls

The interface of the FLORIDA software is divided into five major parts. On top of the window is a menu bar that allows to load and store experimentation settings, if the defaults saved on the last exit are undesired. Below that, is another menu bar that is focused on controlling the external devices. It allows to automatically detect and set up the external devices and take a fresh picture with the camera. Moreover, the currently displayed picture analysis mode—that is raw, threshold or clustering—can be selected here as well as vial detection rendering toggled. The actual image is displayed in the center of the GUI window.

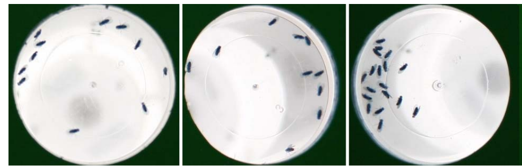
On the right side of the window is a pane with the experimentation settings. Here, the FLORIDA laboratory assistants can configure analysis parameters, like ϵ, θ, ρ, device properties, like the shake time and its lead, as well as result options, like e.g. the path, where the results are stored in a comma-separated value format. On the bottom of the main application window a system log is displayed. It informs the user about the software's status, which includes among other things which devices are connected and which not. The logger can be closed at anytime in order to enlarge the image and can be re-opened via the top menu.

B. Software

The source code of the software can be obtained from the authors' public Github repository [3] and is licensed under BSD-style restrictions, meaning it can be used and modified free of charge. It is written in C++ with the help of the Qt programming framework [1] and the interface has been designed using its built-in creator suite.

In order to communicate with the external devices and to analyze the images the FLORIDA software requires other software dependencies, all of them being open-source as well. For obtaining images from the SLR libgphoto2 [15] is utilized. This library enables camera control automation including image streaming to computers. Supporting a large set of models from common vendors allows a transparent exchange of the actual SLR. The shaker, or electric engine, is remote controlled via an USB relay that is programmed using libusb [9], implementing only the required packages to turn it on or off. Due to the use of a standardized communication protocol for the relay, an exchange of the USB relay is possible.

Internally, the FLORIDA software employs two software packages to process the images. On the one hand, we have OpenCV [4] providing convenience functions for image editing, like color space conversion, cropping, threshold and so forth. On the other hand, a stripped down OpenMP-version of HPDBSCAN [12]—a parallel processing variant of the regular DBSCAN—is used, to fully utilize the computer's processing capabilities. An installation script for the FLORIDA software can be found in the Github repository, effectively obtaining the newest version, compiling it and installing it, including setting correct access rights for the devices.



(a) Samples from data set D_F

(b) Samples from data set D_N

Fig. 5. Samples from two different FLORIDA experiments. While D_F was contains high quality images, the data from D_N differs in terms of illumination and resolution.

C. Hardware

In the current FLORIDA experimentation setup a dual-core laptop with 2.0 GHz and 2 GB of RAM is controlling the other devices. On it is the Fedora 21 operating system installed. The pictures are taken by a Canon EOS 5D Mark I SLR camera, which has a native resolution of 4368×2912 pixels and full frame image sensor format. The high resolution provides enough image quality to record 20 vials concurrently. The vials are shaken by a electric engine that is switched on and off by a "single channel 5V USB Control switch".

VI. Evaluation

An important aspect of each machine learning pipeline is to measure its performance. For the FLORIDA experiment, dedicated validation data sets have been collected and the model parameter were compared to each other by a well defined loss function. The results of these steps can be found in this section.

A. Datasets

To make an evaluation possible, two datasets have been recorded. The first data set D_N is obtained from an accelerated FLORIDA experiment. Instead of capturing one image each 60 seconds, the interval has been decreased to 15 seconds to increase the number of resulting images. At the point when all flies were sedated and after the removal of poor quality images a total number of 1381 vials image were gathered. This dataset still includes low quality pictures in terms of illumination and noise that should not occur in an actual analysis environment. However, we decided to keep this images in the dataset to give machine learning algorithms the chance to be more sensitive to these outliers. The second data set D_F was recorded in the regular one minute interval over 44 minutes total, resulting in a collection of 880 vials. While D_N will be considered as a benchmark dataset to build more robust models, D_F is

TABLE I
GRID SEARCH RANGES—GRAND TOTAL OF 39,900 COMBINATIONS

	Min	Max	Step
Threshold (τ)	90	120	5
Epsilon (ϵ)	2	11	1
MinPts (θ)	1	19	1
PixelsPerFly (ρ)	100	390	10

considered as real experimental data set. All recorded vials have been counted manually and are available on the research repository B2SHARE [2].

B. Parameter search

As a next step, the parameters for the algorithms needed to be found. While the vial size ψ can be easily determined by measuring it on the images, the parameters τ (threshold), ϵ (DBSCAN search radius), θ (DBSCAN density threshold) and ρ (pixels per fly) are more difficult to estimate and are highly dependent on each other.

First, we introduce a user experience parameter set (UEPS) which is simply a set of parameters that have been defined by laboratory assistants, while experimenting with the FLORIDA software and that provide sufficient enough counting accuracy. These parameters are:

$$\text{UEPS} = \begin{bmatrix} \tau \\ \epsilon \\ \theta \\ \rho \end{bmatrix} = \begin{bmatrix} 100 \\ 5 \\ 15 \\ 200 \end{bmatrix} \tag{5}$$

While the UEPS represents just the manual users subjective perception, an automated grid search about a certain range of parameter sets has been performed. With help of the JURECA [17] supercomputer, the grid search over $39,900$ total parameter sets could be computed in parallel. In Table I one can find the parameter ranges that have been searched.

The grid search sorts the parameter sets ascending by the mean squared error (MSE) value on D_F. We also provide the R^2 value, which is another efficient metric to compare the sum of squared errors to the variance of the data.

$$R^2 = 1 - \frac{SSE_f}{V_f} = 1 - \frac{\sum_i (f_i - \hat{f}_i)^2}{\sum_i (f_i - \bar{f})^2} \tag{6}$$

Where f_i is the actual number of flies $\hat{f}_i$ the predicted number of flies and $\bar{f}$ the mean number of flies. This value tends to one, if the model perfectly predicts the desired outcome and is zero, if the prediction is equals to the mean. Additionally, the MSE and R^2 values were computed for D_N. The top five parameter sets with respect to their MSE on D_F can be found in Table II.

C. Interpretation

The gird search shows that small values for ϵ and θ are favorable and that a threshold τ around 90 yields the best results. Moreover, ρ tends to be smaller than initially guessed in the UEPS. Given that the values of ϵ and θ are set below to

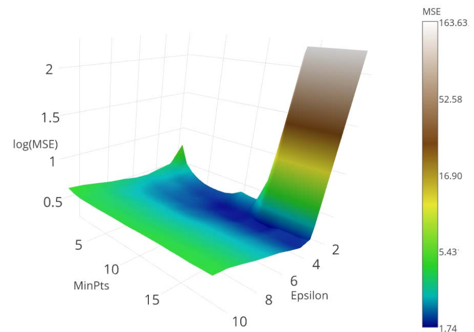


Fig. 6. Three-dimensional projection of the grid search. The DBSCAN parameters ϵ and θ are compared to the resulting MSE. The dark blue regions contain the lowest MSE values.

a certain range the error increases drastically, as one can see in Figure 6.

As expected, the MSE and R^2 values are significantly higher for the D_N data set. A possible explanation is the different illumination in the data, where extreme dark regions could be perceived as large fly clusters, or flies with low contrast would not be recognized as such. In order to handle this problem, one would require a dynamical threshold to binarize the image, depending on its light properties. Interestingly enough, the top five parameter sets with the highest pixels per fly value ρ result in the lowest MSE on D_N. This underlines the statement from above, that large ρ values will result in a lower MSE for accidentally misclassified fly clusters.

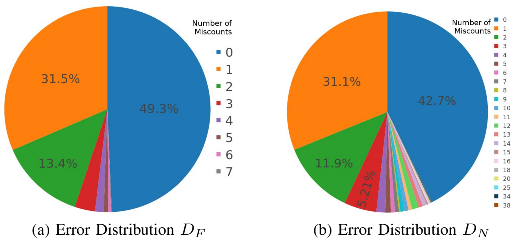
The MSE of 1.745 on D_F is equal to a standard deviation of 1.32 miscounted flies on all images. The results in Table II and the flat grid search surface in Figure 6 indicate that the most optimal parameter sets are similar to one another and robust enough for experimentation conditions. Therefore, we suggest to use the best found parameter set, further referred as grid search parameter set (GSPS) which is:

$$\text{GSPS} = \begin{bmatrix} \tau \\ \epsilon \\ \theta \\ \rho \end{bmatrix} = \begin{bmatrix} 90 \\ 3 \\ 9 \\ 160 \end{bmatrix} \tag{7}$$

Furthermore, in case of deviating experiment environments, new parameters can be found by using the GSPS as a basis.

TABLE II
GRIDSEARCH TOP FIVE RESULTS

#	τ	ϵ	θ	ρ	MSE$_{D_F}$	$R^2_{D_F}$	MSE$_{D_N}$	$R^2_{D_N}$
1	90	3	9	160	1.745	0.946	12.433	0.727
2	90	3	9	150	1.750	0.946	12.912	0.717
3	95	3	15	160	1.757	0.946	19.233	0.578
4	90	3	9	200	1.768	0.946	11.355	0.751
5	90	3	10	160	1.770	0.946	12.559	0.724
UEPS	100	5	15	200	3.325	0.899	26.732	0.414



(a) Error Distribution D_F (b) Error Distribution D_N

Fig. 7. Absolute miscounting errors made by our model, using the grid search parameter set (GSPS).

The MSE performance increase of the GSPS over the initial UEPS is 190% for D_F and 215% for D_N. While this is an significant improvement, the MSE may not provide enough information to validate how accurate the model counts flies. By listing the occurrences of miscounted flies in Figure 7, the reader should get an impression of the model's performance. In the D_F data set, nearly half of the vials has been counted correctly, while another 45% of the vials were miscounted by one to two flies. Only five percent show up higher error rates with up to seven miscounts. The performance in D_N shows that the number of correctly counted flies is lower, but the main drop in performance comes from a low number of extremely high errors. As mentioned before, this could be attributed to dark regions that are misclassified as flies.

In summary the performance of the model exceeds our expectations. Nevertheless, for dark and noisy images a more robust model is required. For well prepared, equally illuminated experiment environments the accuracy of the model is sufficient.

VII. Conclusion

In this paper we presented an automatic image analysis pipeline that allows the counting of intoxicated fruit flies in the FLORIDA assay. We have proposed both, a experimentation setup for devices as well as the analyzing algorithms. The latter was implemented in a GUI application that is currently in productive use at the University of Cologne. In an empirical evaluation of the software we were able to obtain an MSE of 1.745. An adaptation of the method to similar segmentation and counting problems in lab environments is possible through the adjustment of a few model parameters.

VIII. Future Work

More robust models lead to generalization for more scientific domains. One idea is to use the generated fly masks from our model, to train a segmentation convolution neural network as proposed in [19]. With the help of data augmentation, i.e. modifying illumination and rotation, a robust model could be trained, without the need for an explicit threshold parameter tuning. Another approach is to train a convolution neural network through reinforcement learning similar to what Mnih et al. [20] described, in order to dynamically adapt the models parameters to the image. Finally, image classification could be used on the found clusters, to discover multiple flies in one cluster. This step would need the creation of an additional data set, providing images of found clusters and labeled with the number of visible flies. This paper will be followed by biological experiments at the University of Cologne.

References

[1] J. Blanchette and M. Summerfield. *C++ GUI programming with Qt 4*. Prentice Hall Professional, 2006.

[2] C. Bodenstein. Florida sample data. http://hdl.handle.net/11304/29fb9a58-1558-40c0-b3e3-2642b12f8e70. [Data Set].

[3] Bodenstein, Christian and Götz, Markus. FlyDetector Github repository. https://github.com/cbodenst/FlyDetector. [Online, accessed 21-04-2016].

[4] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[5] M. E. Celebi, Y. A. Aslandogan, and P. R. Bergstresser. Mining biomedical images with density-based clustering. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 1, pages 163–168. IEEE, 2005.

[6] C. W. Chen, J. Luo, and K. J. Parker. Image segmentation via adaptive k-mean clustering and knowledge-based morphological operations with biomedical applications. *Image Processing, IEEE Transactions on*, 7(12):1673–1683, 1998.

[7] G. B. Coleman and H. C. Andrews. Image segmentation by clustering. *Proceedings of the IEEE*, 67(5):773–785, 1979.

[8] Duda and Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[9] Erdfelt, J and Drake, D. libusb. http://www.libusb.org/. [Online, accessed 21-04-2016.

[10] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[11] A. Ford and A. Roberts. Colour space conversions. *Westminster University, London*, 1998:1–31, 1998.

[12] M. Götz, C. Bodenstein, and M. Riedel. Hpdbscan: highly parallel dbscan. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 2. ACM, 2015.

[13] J. Han, M. Kamber, and J. Pei. *Data Mining: concepts and techniques - 3rd ed.* Morgan Kaufmann, 2011.

[14] R. M. Haralick and L. G. Shapiro. Image segmentation techniques. *Computer vision, graphics, and image processing*, 29(1):100–132, 1985.

[15] Hubert Figuière. libgphoto2. http://www.gphoto.org/proj/libgphoto2/. [Online, accessed 21-04-2016.

[16] Kettaf, Bi, and Beauville. A comparison study of image segmentation by clustering techniques. In *Signal Processing, 1996., 3rd International Conference on*, volume 2, pages 1280–1283. IEEE, 1996.

[17] D. Krause and P. Thörnig. JURECA: General-purpose supercomputer at Jülich Supercomputing Centre. *Journal of large-scale research facilities*, 2:A62, 2016.

[18] B. N. Li, C. K. Chui, S. Chang, and S. H. Ong. Integrating spatial fuzzy clustering with level set methods for automated medical image segmentation. *Computers in biology and medicine*, 41(1):1–10, 2011.

[19] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] H. Ng, S. Ong, K. Foong, P. Goh, and W. Nowinski. Medical image segmentation using k-means clustering and improved watershed algorithm. In *Image Analysis and Interpretation, 2006 IEEE Southwest Symposium on*, pages 61–65. IEEE, 2006.

[22] Scholz and Mustard. Invertebrate models of alcoholism. In *Behavioral neurobiology of alcohol addiction*, pages 433–457. Springer, 2011.

[23] C.-H. Teh and R. T. Chin. On the detection of dominant points on digital curves. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(8):859–872, 1989.

[24] Q. Ye, W. Gao, and W. Zeng. Color image segmentation using density-based clustering. In *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, volume 3, pages III–345. IEEE, 2003.

Appendix D

Paper IV

Noname manuscript No.
(will be inserted by the editor)

Automatic Water Mixing Event Identification in the Koljö Fjord Observatory Data

Markus Götz · Mikhail Kononets · Christian Bodenstein · Morris Riedel · Matthias Book · Olafur Petur Palsson

Received: July 31, 2017/Accepted:

Abstract This study addresses the task of automatically identifying water mixing events in the multivariate time series of salinity, temperature and dissolved oxygen provided by the Koljö fjord observatory. The observatory is used to test new underwater sensory technology and to monitor water quality with respect to hypoxia and oxygenation in the fjord, and has been collecting data since April 2011. The fjord water properties change, manifesting as peaks or drops of dissolved oxygen, salinity and temperature, when affected by inflows of new water originating from the open sea or by rivers connected to the fjord system. An acute state of oxygen depletion can harm wildlife and the ecosystem permanently. The major challenge for the analysis is that the water property changes are marked by highly varying peak strength and correlation between the signals.

The proposed data-driven analysis method extends existing univariate outlier detection approaches, based on clustering techniques, to identify the water mixing events. It incorporates three major steps: 1. smoothing of the input data, to counter noise, 2. individual outlier detection within the separate variables, 3. clustering of the results using the DBSCAN clustering algorithm to determine the anomalous events. The proposed approach is able to detect the water mixing events with a *F1*-measure of 0.885, a *precision* of 0.931—that is 93.1% of all events have been correctly detected—and a *recall* of 0.843—that is 84.3% of events that should have been found actually also have been. Using the proposed method the oceanographers can be informed automatically about the status of the fjord without manual interaction or physical presence at the experiment site.

M. Götz · C. Bodenstein · M. Riedel
Juelich Supercomputing Center, Research Center Juelich,
Wilhelm-Johnen-Straße, 52428 Jülich, Germany,
E-mail: {m.goetz, c.bodenstein, m.riedel}@fz-juelich.de
ORCiD: 0000-0002-2233-1041

M. Kononets
Department of Marine Science, University of Gothenburg,
Box 461, SE-405 30 Göteborg, Sweden,
E-mail: mikhail.kononets@marine.gu.se

M. Götz · C. Bodenstein · M. Riedel · M. Book · O. P. Palsson
Faculty for Industrial Engineering, Mechanical Engineering
and Computer Science, University of Iceland,
Sæmundargötu 2, 101 Reykjavik, Iceland, E-mail: {book, opp}@hi.is

Keywords Multivariate Time Series Analysis, Koljö Fjord Observatory, Water Mixing Event Detection, Clustering, DBSCAN

Acknowledgments

The installation of the Koljö fjord cabled observatory was carried out by the University of Gothenburg in collaboration with MARUM, University of Bremen, Germany, and funded by the European Commission projects ESONET-NoE (contract number 036851), HYPOX (grant agreement number 226213) and EMSO (grant agreement number 211816). This work is also supported by Aanderaa Data Instruments AS providing the Doppler Current Profiler instruments, other material and financial support to run the Koljö fjord observatory.

1 Introduction

Many important events can be observed using underwater sensor systems. Algal blooms, various water mixing and renewal events appear on the background of seasonal changes. Salinity and temperature are essential

parameters as they determine seawater density and thus circulation. Chemical variables, like oxygen or chlorophyll, can serve as indicators of ongoing biogeochemical processes as well as chemical markers of different water masses and the 'health' of the waterway. Current sensors directly measure water transport. One needs to observe many parameters at the same time to be able to correctly detect and interpret changes as ongoing events, and one needs to have experience in the observations as well as take into account hydrography. Often several processes occur at the same time, and different parameters also change with relation to other signals like sunlight intensity, wind and so forth. This complex event detection task can be delegated to an automated system. One goal of this work is to develop a data analysis system that can be integrated into existing monitoring infrastructures to provide automated event detection and warning system functionality. Potential applications can be numerous—real-time detection of various natural and anthropogenic events, e.g., algal blooms, water mixing events, or leakages from underwater oil pipelines based on analyses of multivariate sensor data.

Outlier detection, sometimes also called anomaly or novelty detection, is the process of finding observations within data that are significantly different from the remainder of the data set. It can be considered as a special form of classification problems, where one categorizes data items as "anomalous" or "normal" based on their values. There are various application fields for outlier detection, from flow control monitoring in oil pipelines to credit card fraud detection. In this work we have applied this method to multivariate oceanographic time series data recorded in Koljö fjord.

The specific research question is given by periods in time when new water from the ocean or the connected fjords comes into Koljö fjord and changes its water properties. A water mixing event is characterized by a sufficient change in sea water temperature, salinity and oxygen, which occurs simultaneously and for the same time period. The changes in the water properties may not be evenly distributed among the different variables. For this reasons, simple rule-based or peak identification approaches are not sufficient.

The major analysis goal is to determine whether the influent waters are potentially harmful to flora and fauna or heavily change the water chemistry, damaging the natural reserve of the fjord. This also gives the oceanographers and the local government of Borhus county the possibility to coming to well-grounded decisions whether environmental engineering efforts need to be taken. These could for example include restrictions on fishing or ecological policies, like the reduction of fertilizer use around the fjord. For the application domain scientists it is going to save labor and cost as they do not have to manually monitor the data as it is recorded. In the bigger picture, this will also help to better understand the hydrologic cycles on our planet and their impact on Earth's climate.

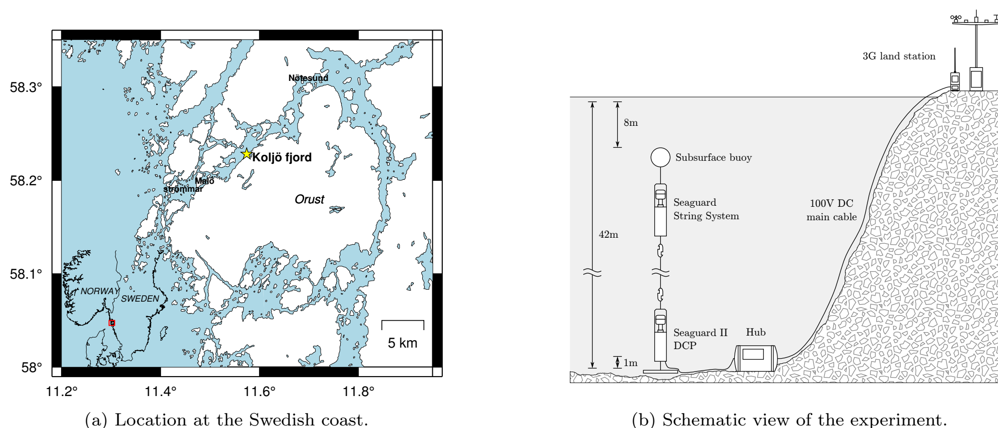
The scientific literature proposes a number of techniques for the (semi-) automatic identification of outliers that are harder to identify. These include, among others, supervised machine learning, such as support vector machines [8], autoregressive–moving-average (ARMA) [39] model construction and subsequent deviation identification, statistical hypothesis testing based on the assumed underlying data distribution, clustering methods and others. In this work we extend existing univariate data analysis approaches of the latter category, by utilizing the DBSCAN clustering algorithm to determine the water mixing events in a multivariate time series. Our main reason for selecting this methodology was the lack of precise labels per sample, preventing the use of potentially more robust supervised methods. One of the aims of the Koljö fjord observatory is to perform monitoring with high temporal resolution, that helps to resolve event dynamics. However, from the data analysis point of view, this makes it more difficult to identify the starting and end point of each event precisely. This is why we have decided to use the DBSCAN clustering algorithm together with a simplified representation of each water mixing event by its central point. This approach is explained in Section 6.1, where the validation of the analysis results is discussed.

The remaining sections are organized as follows. Sections 2 and 3 introduce the background for this work and survey related studies. The Koljö fjord observatory data and its properties are explained in Section 4. Subsequently, Section 5 details the proposed analysis approach. An evaluation of its effectiveness is given in Section 6, before the paper is concluded in Section 7.

2 Background

2.1 The Koljö Fjord Observatory

The Koljö fjord observatory [23] is situated on the Swedish western coast. Its primary purpose is real-time monitoring of water mixing with regard to semi-permanent and seasonal hypoxia, specific to many fjord systems, using the Koljö fjord as an example. At the same time new measurement hardware is often tested, as Koljö fjord provides a very wide range of conditions—from semi-permanent anoxia in deep water to very well oxygenated and extremely productive



(a) Location at the Swedish coast.

(b) Schematic view of the experiment.

Fig. 1: The Koljö fjord observatory.

surface waters. Koljö fjord is very well protected with regard to wind and waves and is easily accessible by research vessels from Kristineberg field station run by the University of Gothenburg. A practical side-effect of the observatory work is that the residents around the fjord have a possibility to inform themselves about the general water quality and effects such as seasonal hypoxia and algal blooms.

The Koljö fjord is a part of the Orust fjord system around the islands of Orust and Tjörn on the Swedish west coast. It is connected to the Skagerrak through the narrow and shallow Malo Strömmar straight (9 m deep) at its south-west end and to the Havstenfjord through the Nötesund sill (12 m deep) at the north-east side [17]. Above the sill depth the Koljö fjord is characterized by free water exchange. Surface water properties resemble the conditions found in Kattegat and change rapidly [4]. Below the sill depth, the residence time of the basin water is long—about three to four years—as water renewal is restricted to rare occasions of exceptional weather conditions, e.g. persistent easterly winds. As a result, the basin water is anoxic most of the time [33]. The Koljö fjord is an example of the particularities of many Scandinavian fjords and an interesting research site in terms of biogeochemical activity. Due to its well-protected location, highly variable and dynamic conditions, and strong seasonality, the fjord is well suited for field testing of new technologies, with regard to sensor measurements as well as data interpretation and analysis.

A cabled observatory was installed in the Koljö fjord in April 2011 at $58°22'82.5''N$, $11°57'40.0''E$ in 42 m depth, and has been operational since then. Figure 1a shows its position on the map. The observatory consists of a land station, an underwater hub and experimental nodes. A main cable connects the land station with the hub, powering it and transferring data, allowing for attachment of up to four experimental nodes. The experimental node currently in use has a Doppler Current Profiler (DCP), model Recording DCP-600 (from 2011 to 2014) [1] or Seaguard II DCP (since 2015) [2] installed about 1 m above ground, and a Seaguard String System [3]. The DCPs measure currents with a resolution of 1 m through the water column and are also equipped with sensors to measure temperature, salinity, pressure, and oxygen above the bottom. The Seaguard String System collects additional oxygen, conductivity and temperature readings from around 30 sensors at several depths between 27 m and 8 m below the surface.

Dissolved oxygen and temperature measurements have been made using oxygen optodes model 4835. The optodes are stable, with a field precision of about 0.2 µM and an absolute accuracy of about 2% [6]. Conductivity, salinity and temperature measurements have been collected using the sensors model 4319 with an absolute accuracy of 0.05 $\frac{\text{mS}}{\text{cm}}$ for conductivity and 0.05 °C for temperature. Figure 1b shows a schematic view of the observatory experimental setup that represents a real world data source used in the analysis.

2.2 DBSCAN Algorithm

Clustering algorithms in the field of data-mining or machine learning are used to aggregate similar objects into common groups. The *Density-Based Spatial Clustering for Applications with Noise* algorithm (DBSCAN) is a particular, density-based clustering algorithm that was published by Ester et al [10]. Its principal idea is to

find dense areas and to expand these recursively in order to find clusters. A dense region is thereby formed by a point that has at least *minPoints* neighboring points within a given search radius ε. This point with the surrounding dense area is also called the *core* of a cluster. All other neighboring points are considered border points of the cluster. For each of them the density criterion is recursively reapplied. Given that it is fulfilled, the cluster is consequently expanded, turning the currently checked point into a *core* point and its ε neighborhood into border points. Points of the same cluster that are less than ε distance units apart from each other, are called directly-density-reachable (DDR). All others that are further away, but are transitively directly-density-reachable are called density-reachable (DR). The latter relationship is not symmetric, because a *border* point must not be DR by a *core* point, however, the opposite holds true. Therefore, all points within a cluster are called density-connected (DC) if either one of them is DR by the respective other.

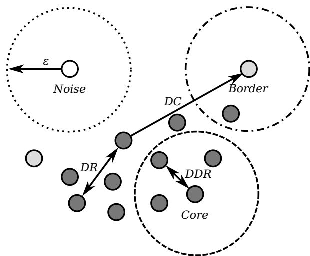


Fig. 2: Exemplary DBSCAN clustering with $minPoints = 4$. Cluster core points are dark gray, border points light gray and noise white.

All points that do not form a cluster core and that are not "absorbed" through recursive expansion of any cluster are regarded as *noise* that form their own special cluster set. A formal definition of the algorithm can be found in the publication of Ester et al [10]. The major advantage of DBSCAN, compared to other traditional clustering algorithms such as K-Means [28] or hierarchical clustering [16], is that it is able to detect arbitrarily shaped clusters without having to know the number of clusters a priori. In addition to that, it is able to efficiently detect outliers through its notion of noise. Figure 2 shows an example of a DBSCAN clustering.

3 Related Work

Clustering algorithms have been successfully used in the recent decade to analyze time series data. Liao [27], for example, provides an in-depth survey of methods, algorithms and use cases for time series data clustering with a strong emphasis on explaining how time series data can be preprocessed, and how features can be extracted in order to make it susceptible to standard clustering algorithms. Similar to that, Hodge and Austin [19] propose approaches to detect outliers in time series data using various techniques, clustering being a major one.

Less recent application examples are presented by Goutte et al [14] and Himberg et al [18]. Both use cases come from the field of neuro-sciences, in which they analyze fMRI and MEG image time series data to detect voxel activations or independent brain areas respectively. Bagnall and Janacek [7] explain in their publication how to improve computation time in a similar application through time series subdivision. All of the mentioned publications share two common points. First, they try to analyze data that has no or only weak labels, prohibiting traditional classification algorithms, such as the data we are dealing here with and second, they all utilize the K-Means [28] clustering algorithm, or variants of it such as K-Medoids [16].

Jiang et al [20] argue that K-Means is a suitable algorithm for outlier detection, given that the number of clusters is known or can be easily inferred. In their gene expression time series outlier detection use case, similar to the water mixing events, this is not the case. Therefore, they propose to use a modified version of the density-based clustering algorithm DBSCAN, which does not require the cluster count a priori. Their main idea is to perform iterative density-based clusterings that are subsequently ordered in a hierarchical tree based on their inclusion or intersection. Everything below or respectively above a certain density threshold is considered to be data outliers and can either be discarded or further investigated. Jiang et al [20] use the laid out algorithm to identify unusual patterns in univariate serum stimulation time series.

It is worth noting that even the baseline DBSCAN algorithm can be used to detect anomalies in time series data. Pavlidis et al [34] have demonstrated this by predicting drops and peaks in the German mark to US dollar exchange rate using DBSCAN input series clustering. For this, they cluster the univariate time series and based on the resulting clusters, subdivide the data into normal (cluster) and abnormal episodes (noise) as local predictor labels for a subsequent artificial neural network training. Another example is the air space monitoring use case of Gariel et al [12], in

which they automatically detect deviations from the univariate, day-normal airplane trajectories time series in order to report unusual flight paths. Finally, Kut and Birant [26] have proposed their own DBSCAN modifications for the analysis of spatio-temporal data. They cluster the temporal and spatial variables of a dataset independently of one another in order to identify unusually strong waves, similar to what is proposed in our work, except that they do not correlate the independent clusters directly but rather through an elaborate tree search.

In the literature one can also find some existing work on water mixing event detection, or in other words, anomaly detection of water properties. However, these mainly revolve around the automatic analysis of drinking water sources and their potential contamination with pathogens, chemicals and so forth, to ensure potability. Even though their water systems are different from a natural fjord system, in the sense of a much more closed and stable system, the base analysis problem is similar to the one in our work. Zhao et al [40] for instance present a review of various event detection mechanisms. These include among others artificial-neural-network-based detection as presented by Perelman et al [35] or k-nearest-neighbor classification as used in the CANARY system of the United States Environmental Protection Agency [32]. They have the advantage of having very precise ground-truth data for the particular events, which is not available for the Koljö fjord observatory data. In such cases the analysis can be performed using unsupervised clustering methods, as presented by Klise and McKenna [22] and their K-Means analysis. As stated before, DBSCAN is a much better fit for an unknown number of events. This is why this work's method is based on the latter algorithm. To the best of our knowledge our analysis is the first to study water mixing events in non-controlled, therefore much more dynamic, natural environments. With this it has effectively introduced data-driven water mixing event detection models in oceanography.

4 The Observatory Data

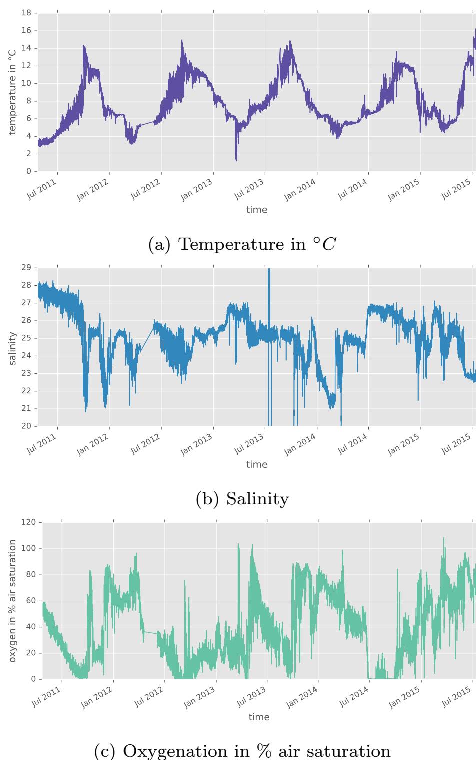
Water conditions in Koljö fjord can be schematically divided into three groups. The surface layer—down to a depth of about 10 m—has the lowest salinity, as it receives fresh water from rivers. In summer, surface waters are well illuminated by the sun, and as a result, algae and animals can settle and grow on any available surface, including the sensors. This process is called biofouling. Sensor measurements in the surface layer require serious antifouling protection, otherwise sensor readings become affected by biofouling during spring, summer and fall. The onset of biofouling is hard to recognize in practice, before a week or more has passed from its start due to its slow, gradual development. Otherwise, surface waters are usually very well mixed by wind, and conditions with regard to oxygen and salinity are stable.

In contrast to that, deep waters (deeper than about 20 m) have the highest salinity and are stagnant most of the time. Due to the fjord system's hydrography and good protection from wind, deep waters cannot practically be affected by wind mixing. Water mixing occurs seldom in the deep waters (once every 3-5 years). They normally contain no oxygen for large stretches of time, so measurement data shows very few characteristics.

The intermediate layer (between 10 m and 20 m) is quite dynamic. This layer can be affected by wind mixing directly and indirectly, including wind-induced water movements in and out of the fjord through the Nötesund strait. Strong changes in all water parameters can occur on the time scale of days and hours. In the intermediate depths the sensors are very seldom affected by biofouling processes due to scarce sunlight for algae and low oxygen for animals. For this reason, the temperature, salinity and oxygen data measured at 15 m depth have been selected.

Specifically, the optode s/n 29 and the conductivity sensor s/n 242 deployed at 15 m water depth are considered. The data has been collected in the time period from April 19th, 2011 16:42:01 UTC to July 15th, 2015 09:09:12 UTC and includes a grand total of 71,350 data points, excluding missing values, that have been sampled every $30min$. The full observatory dataset contains numerous water parameters measured at different depths as well as additional information about the status of the observatory sensor system, such as for example their three-dimensional orientation, power supply and so forth. The additional data is of utmost importance. It is used for quality checking of raw sensor readings and understanding which changes must be interpreted as water mixing events. However, after one has marked the events, the additional data is no longer important for the further analysis, explaining the limitation to the actually used variables time, temperature, salinity, and oxygen saturation. The raw data for the investigation can be found on the observatory webpage [23] as well as on the open earth science data repository PANGAEA [9, 24]. Figure 3 depicts the data curves before any preprocessing has taken place, but with gaps linearly connected (e.g. in May 2012).

Due to sensor maintenance, calibrations and outages the data set contains gaps—i.e. samples with a time difference larger then 30 min. Table 1 lists them in detail. Note, that occurrences of oxygen supersaturation—e.g.



(a) Temperature in $°C$

(b) Salinity

(c) Oxygenation in % air saturation

Fig. 3: Raw data signals as obtained by the Koljö fjord observatory sensors.

in the spring months—are normal, i.e., not erroneous readings and can be explained by algal bloom and their photosynthesis.

As part of the data preprocessing, these gaps are filled by linearly interpolating between the last value before and the first value after the gap, sampling every 30 min from the beginning. This may result in an extra data point at the end of each interpolation interval, which has a smaller time difference than 30 min, to the subsequent point, but which should not affect the analysis. After the interpolation the dataset has a grand total of 74 299 samples. It can be obtained from the scientific data repository B2SHARE [25].

5 Analysis Process

5.1 Moving-Average Smoothing

As can be seen in Section 4, the data signals are clearly noisy, with strong point-to-point variations. For the analysis goals, one only requires knowledge about the general outline of the curve—i.e. major peaks or drops. In fact, singular strong outlying data points can hinder the automated water mixing detection by presenting the analysis process with a lot of short, but strong consecutive peaks disguising themselves as short lived water mixing events. This naturally means that the analysis must first suppress the noisy episodes.

Table 1: Gaps due to missing samples in the raw data of the observatory.

Start	End	Missing Samples
2011-09-29 12:12:01	2011-09-29 20:41:54	16
2012-04-18 12:41:54	2012-06-04 19:13:01	2269
2012-06-27 15:43:01	2012-06-27 20:09:57	8
2012-08-29 13:39:57	2012-08-29 18:09:57	9
2012-10-25 10:09:57	2012-10-25 11:09:57	2
2012-10-25 13:09:57	2012-10-25 17:50:13	9
2013-04-24 09:50:13	2013-04-25 15:30:40	59
2013-06-03 11:30:40	2013-06-05 16:11:38	105
2013-08-15 12:11:38	2013-08-23 16:18:40	392
2013-12-12 10:18:40	2013-12-12 17:35:08	14
2014-03-05 10:35:08	2014-03-05 14:13:20	7
2014-06-04 08:13:20	2014-06-04 16:00:45	15
2014-09-01 11:00:45	2014-09-01 17:17:44	12
2015-02-11 09:47:44	2015-02-11 17:56:05	16
2015-04-07 09:56:05	2015-04-07 18:09:12	16

A typical approach in signal processing to tackle this problem is to smooth the time series data by applying a *windowed filter* to each point in the datasets. Since the whole analysis pipeline shall be later used to predict water mixing events in real-time, one cannot utilize filtering methods that require any data points beyond the current value—or in other words "future" values. At the same time, there is no argument in favor of any particular weighting scheme, such as triangular or exponential windows, as the water mixing events vary heavily in their characteristic shape.

Therefore, a standard median filter [5] with a window size of three is employed first, to correct singular runaway values. These are caused by misreads or external objects, interference with sensors, and can be seen in the salinity curve of Figure 3 in July 2013 for example. A larger filter window size than three is undesired as it introduces a systematic lag in the smoothed data due to the medians statistic robustness[1].

Afterwards a simple moving average (SMA) [21] provides the actual smoothing with sensitivity to slope changes. Having removed the strong noise, the distribution is statistically stable enough to justify using the mean, which in turn is more sensitive to fluctuations in the signal (characteristic for the mixing events) compared to a larger-windowed-median filter. Equations (1) and (2) formalize this procedure with x_t being a sample

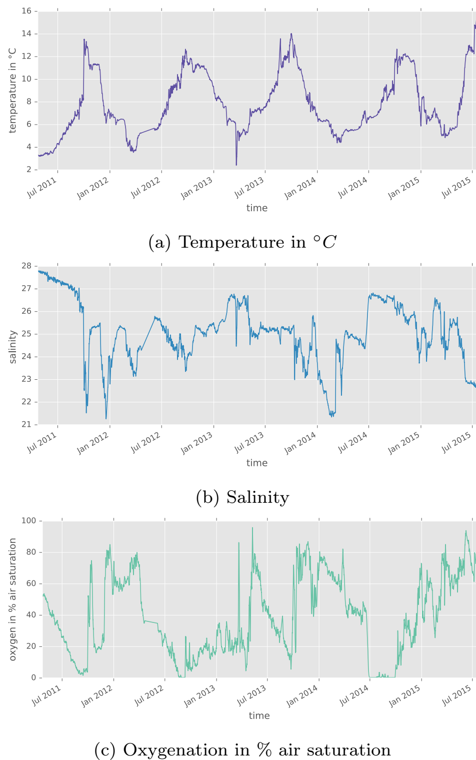
[1] Median-smoothed curves reflect slope changes with a delay of half the filter window size.

from the dataset, w, the SMA window size, and x_{t-n}, a sample n steps back in time (as defined by Madsen [29] for example).

$$x_{\text{MED},t} = \text{median}(x_{t-1}, x_t, x_{t+1}) \tag{1}$$

$$x_{\text{SMA,t}} = \frac{\sum_{i=0}^{w-1} x_{\text{MED,t-i}}}{w} \tag{2}$$

Figure 4 shows the smoothed time series for all signals. Due to space considerations, only small plots of the curves are presented here, but magnifiable versions can be found on the publicly available, scientific data repository B2SHARE [15]. For the first $w - 1$ data points of each signal, a smoothed value cannot be computed, as the number of previous values is insufficient. The analysis simply omits these values, as they constitute a maximum of just two and a half days.



(a) Temperature in $°C$

(b) Salinity

(c) Oxygenation in % air saturation

Fig. 4: Smoothed time series with a median and simple moving average filter using the exemplary window size $w = 96$, omitting the first two days.

5.2 Univariate Outliers

For the second analysis step, one needs to solve the problem of detecting outliers, i.e., peaks within a single data signal, so that these can later be used to find the global water mixing events across multiple variables. As previously explained, outliers—i.e. the water mixing events—are characterized by strong peaks or drops of the signal. To reveal these variances, the time step difference, or the gradient, is computed. An outlier can then be found by defining a threshold, beyond which the gradient is too strong to be a regular data point. A statistically sound value for the threshold is a multiple of the standard deviations, as one can expect the remaining data signal to be more or less steady. Figure 5 shows the one-step time difference of the curves, and the one, two and three standard deviations levels. Formally, an outlier within a single signal can be defined as follows, with X being the respective signal, c, the multiple of the standard deviation σ, and d the time-step difference ($d = 1$ for a simple gradient).

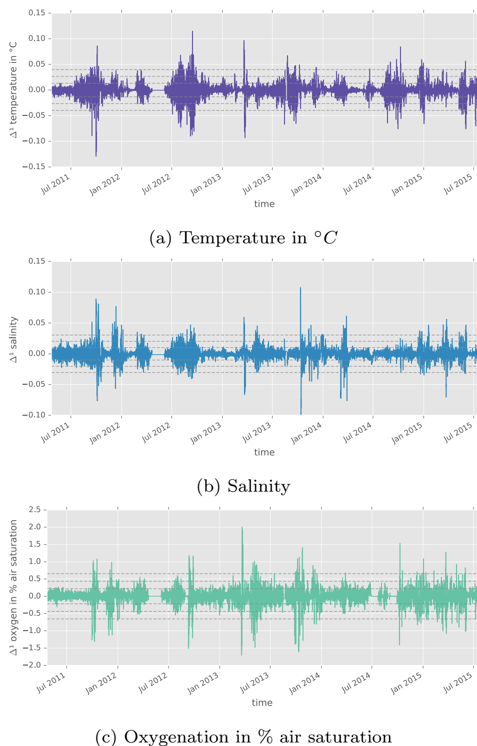
$$\Delta^d x_t = \begin{cases} x_t - x_{t-d} & \text{for } d \leq t \\ 0 & \text{else} \end{cases} \tag{3}$$

$$Outliers_X = \{t \mid \forall x_t \in X : abs(\Delta^d x_t) > c * \sigma(X)\} \tag{4}$$

5.3 Water Mixing Event Detection Using DBSCAN

In order to find the water mixing events, the individual outliers of each of the signals have to be correlated. Defining a threshold per signal for certain time periods and subsequent voting on whether it is a mixing event or not is not sufficient, because not all of the signals will spike equally strong. Instead, the density of outliers of all signals combined must exceed a certain threshold, independent of the share of the individual variables. An established technique to tackle this problem is clustering—i.e. unsupervised detection of groups in data. For this application, a clustering algorithm is required that is able to detect an arbitrary number of clusters based on sample density. As explained in Section 2.2, DBSCAN fits these requirements perfectly. Each of the resulting groups of the clustering represent a singular water mixing event.

The Equations (5) and (6) formalize the explained process with $Signals$ being the set $\{Temperature, Salinity, Oxygen\}$. Figure 6 presents the result of such a DBSCAN clustering. It displays the outliers of the individual scales horizontally for each signal respectively



(a) Temperature in $°C$

(b) Salinity

(c) Oxygenation in % air saturation

Fig. 5: One step difference of the signals, including one, two and three standard deviation levels as dashed lines.

and in red the found clusters for the exemplary parameters, evaluated through system grid search (see Section 6.2) with a standard deviation level c=1 and a sample density $minPoints$=48 within the search radius ε=22 ∗ 30 min. One can also see that not all individual signals need to have outliers at a given period in time in order to form a cluster, as for example in September 2012, for oxygenation, or in March 2013, for salinity.

$$Outliers = \bigcup_{X \in Signals} Outliers_X \quad (5)$$

$$Events,\ Noise = DBSCAN(Outliers, minPoints, \varepsilon) \quad (6)$$

6 Evaluation

In order to evaluate the performance of the proposed algorithm, it is necessary to verify it based on real world data. For this, the data, introduced in Section 4, is utilized as well as the ground truth presented in this

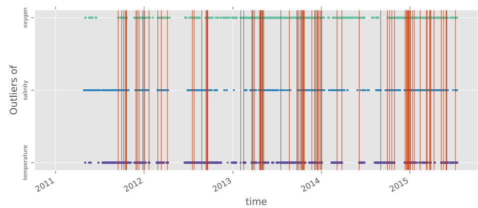


Fig. 6: Outlier detection results for each data signal with outliers as dots and found multi-variate clusters as red bars. The exemplary parameters for this analysis are $w = 96$, $c = 1$, $d = 1$, $\varepsilon = 22$ and $minPoints = 42$ found via grid search.

section. First, one needs to find a suitable parameter set that can explain the data well, based on a performance measure, and subsequently test it on unseen data. The traditional classifier construction involves a training, validation and test set. The former is not applicable, though, since there are no intrinsic parameters, say weight matrices, in this method that have to be learned. Therefore, the models will be directly validated on the validation data and verified on the test data. The validation set consists of the first two-thirds of the entire data based on the time scale, and the test data is the last third.

6.1 Ground truth

The data has been manually annotated by domain experts with the points in time at which water mixing events took place. This has proven to be a time-intensive process, requiring to establish a common understanding of terms and definitions, as well as a substantial learning effort on the data analysis experts' side in understanding the application domain better. Especially the identification of particular events could essentially only be done by the domain experts due to the complex nature of the oceanic system. The final 69 identified events, which is less than 1% of the total samples, each constitute a single date and time of day representing the approximate center point of a water mixing event. The nature of these labels add to the difficulty of the analysis problem. They are fuzzy at best, with a varying confidence even among the domain scientists whether a certain part of the time series actually marks a water mixing event or not. Therefore, we have only included the ones the domain scientists have all agreed upon. In addition to that, they are also low in count, effectively making pattern detection generalization hard. At the same time, the lack or precise per-

sample labels prevents the out-of-the-box utilization of traditional classification methods.

Moreover, nine out of these events have to be taken with reservations as they look like water mixing events but are not in reality. They can be attributed to erroneous reads of the sensors due to biofouling processes from December 2013 to February 2014 (seven events) and sensor re-deployments in September 2014. For the sake of evaluation simplicity, they are included as regular events. A list of the water mixing events and an additional binary flag indicating whether it was faulty or not can be found on the open scientific data repository B2SHARE [25]. Figure 7 shows the events on top of the smoothed data curves.

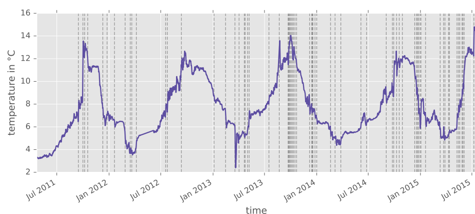


Fig. 7: Water mixing events ground truth projected on top of the temperature signal. Each event is presented as a single, vertical, dashed line in gray.

The interpretation of the selected dataset's events belonging to the ground truth was mainly based on the readings in the fjord and has been verified using data measured outside of it. For salinity, temperature and oxygen measurements, the monthly sampled, nation-wide run monitoring data of the Sveriges meteorologiska och hydrologiska intitut [37] (SMHI, engl: Swedish meteorological and hydrological institute) have been used. Their measuring site is located a few hundred meters away from the observatory position. As for verifying the currents, temperature and oxygenation, an indirect approach has been selected. Wind and sunlight data assimilated from a weather station run by the University of Gothenburg at the Kristineberg field station [38] has been correlated with the water's movement data of the observatory and to assisted in its verification. The weather station is situated about eight kilometers away from the observatory.

6.2 Grid Search

In order to optimize the water mixing event detection rate, a systematic grid search of the five model parameters has been performed—the smoothing filter size w, time step difference d, standard deviation level c, DBSCAN search radius ε and point density *minPoints*. The boundary-inclusive search space is displayed in Table 2 and spans a total of $313,200$ model variants.

Table 2: Search parameter grid, bounds are inclusive.

Parameter	Lower bound	Upper bound	Step
Filter size w	24	120	12
Step difference d	1	3	1
Standard deviations c	0.25	2	0.25
Search radius ε	12	60	2
Point density *minPoints*	6	120	2

As a performance measure for the models the, F_1-score [36] is used—i.e. the harmonic mean of the precision and recall of the detected mixing events, compared to the ground truth presented in the above Section 6.1. This essentially balances the detected *true positives* with the *false-positives* and *false-negatives.*

$$F1 = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{7}$$

Defining the terms true positives and false negatives is not straight-forward as the computed clusters are sets of points, whereas the ground truth data is punctual. According to our definition, an event is considered as detected or a true-positive if the event center point lies within a cluster's time limits plus/minus a tolerance of two days[2]. In every other case the predicted event is a false-positive.

Figure 8 presents the resulting F_1-score surface plots for the best models with the same ε and *minPoints* values, but varying c, w and d parameters in the validation phase. It shows a certain range of well performing ε and *minPoints* values as "crests" on the score surface. Table 3 shows the top ten candidates from the validation phase model construction, with the best-performing marked in bold. A number of observations can be made about the results.

First, the moving average filter size w is equal to either 96 or 120 for all but one of the top models and dozens following not displayed here. This makes sense when one considering the natural phenomena and behaviors of the domain problem. The tidal cycle lasts 12.4 h, or in other words 24 to 25 sample points. Since the tidal cycle introduces harmonic oscillations, the filter window size must be proportional to the tide cycle length. This way, the smoothing algorithms will be able to compensate for the periodical variations in the most efficient way. Moreover, smoothing out 96, respectively 120, samples indicates that one wants to observe strong

[2] Estimated minimum duration of a typical water mixing event.

Table 3: Top ten models on the validation data

Rank	ε	m	w	d	c	$F1$	precision	recall
1	**22**	**42**	**96**	**1**	**1.00**	0.8070	0.7666	0.8518
2	16	32	120	2	1.25	0.7967	0.7538	0.8448
3	14	28	120	3	1.50	0.7964	0.7758	0.8181
4	12	24	120	2	1.50	0.7936	0.7575	0.8333
5	16	32	84	3	1.00	0.7933	0.7500	0.8421
6	16	32	120	1	1.25	0.7931	0.7666	0.8214
7	14	28	96	2	1.50	0.7920	0.8163	0.7692
8	14	26	120	1	1.50	0.7906	0.7500	0.8360
9	22	40	96	1	1.00	0.7903	0.7101	0.8909
10	22	42	96	2	1.00	0.7899	0.7230	0.8703

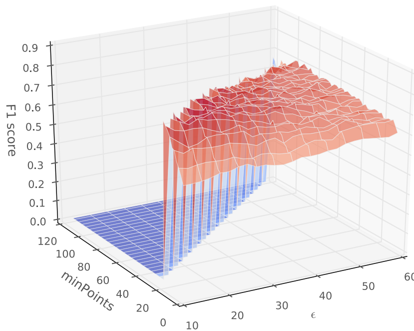


Fig. 8: Validation phase F_1-score surface plot.

variations in the data over the time scale of days, in order to consider it a water mixing event. This corresponds to the fact that water mixing events usually last a number of days. Therefore, a filter size of 96, respectively 120 samples or two to two and a half days seems to be the best trade-off between appropriate noise suppression and information loss.

Second, the standard deviation level c is around 1.0 to 1.5 throughout all the best performing models, suggesting that already small variations in the water properties are indicative of water mixing events, and not only singular strong ones.

Finally, the outlier search radius ε lies within the interval of 12 to 24 samples for the best-performing models. This is identical to a search diameter of more or less 24 to 48 samples or simply half a day to a full one. At the same time, the *minPoints* parameter is mostly within the range of 26 to 42 samples. If one were to take the center points of the intervals as boundaries, this means that $\frac{34}{36} \approx 94.4\%$ of the measurement time points have to have an anomaly within one signal, or if evenly distributed among all of the three signals at hand, $\frac{94.4\%}{3} \approx 31.5\%$ of all data points have to be univariate outliers, in order to detect a water mixing event cluster core.

Table 4: Top models on the test data and placement of the best model of the validation phase in bold.

Rank	ε	m	w	d	c	$F1$	precision	recall
1	24	48	96	1	0.75	0.9687	0.9687	0.9687
2	22	44	96	1	0.75	0.9577	0.9444	0.9714
3	24	42	96	3	1.00	0.9411	0.9411	0.9411
4	12	12	96	2	2.00	0.9411	0.9090	0.9756
5	12	12	96	1	2.00	0.9411	0.9090	0.9756
6	30	48	96	2	1.00	0.9393	0.9393	0.9393
7	16	12	96	1	2.00	0.9382	0.9047	0.9743
8	14	12	96	1	2.00	0.9382	0.9047	0.9743
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
694	**22**	**42**	**96**	**1**	**1.00**	**0.8852**	**0.9310**	**0.8437**

For the test phase, not only the top-performing model from the validation phase has been tested, but all of them, in order to be able to judge the generalization of the approach. Figure 9 shows the resulting F_1 surface. One can clearly see the similar shape and optimal parameter ranges compared to the validation data models. Interestingly, the resulting F_1 scores are slightly better on the test data alone, as can be seen in Table 4. This can be attributed to the more apparent regularity of the water mixing events in the signals.

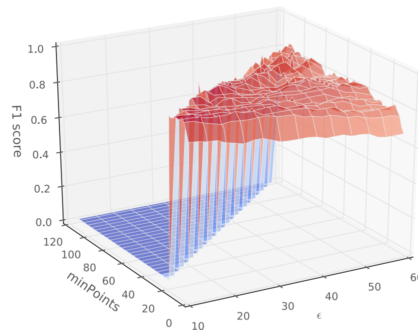
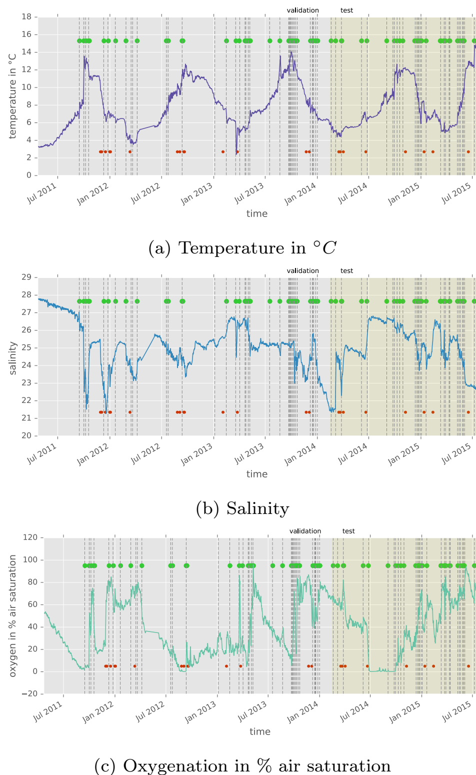


Fig. 9: Test phase F_1-score surface plot.

The chosen model from the validation phase is ranked 694th out of the total 313 200 model candidates. With an 0.06 lower F_1 score, compared to the top placed test model, there is a slight gap. However, it generalizes well enough, being placed in the top 0.22% of all the tested models and is a robust model for production use in the online analysis of new observatory data. Figure 10 shows the final predictions of the top-performing model from the validation phase on both the validation and test data.

The proposed analysis pipeline is able to detect water mixing events in the Koljö fjord with high preci-



(a) Temperature in $°C$

(b) Salinity

(c) Oxygenation in % air saturation

Fig. 10: Predictions of the top-performing model with the parameters $w = 96$, $c = 1$, $d = 1$, $\varepsilon = 22$ and $minPoints = 42$. Ground-truth water mixing events are shown as dashed lines, true-positives as green on top and false-positives as red dots on the bottom.

sion. The false-positive rate is low and constitutes only a fraction of the actual water mixing events. This makes it suitable for productive use as an automatic warning system for water mixing events for the domain scientists at the University of Gothenburg, and an effective monitoring tool for the health of the flora and fauna of the fjord.

6.3 Implementation

The analysis pipeline as well as the grid search are implemented in form of a Python script. It is available on the authors' source code repository [30] for reproducibility. It requires a number of software dependencies that do not come pre-installed with a standard CPython distribution. Following is the list of packages and their respective versions: `mpi4py 1.3.1`, `bokeh 0.11.1`, `numpy 1.11.0`, `pandas 0.18.1`, `scipy 0.17.1` [31]. In addition to that, the analysis employs the C++ OpenMP implementation of HPDBSCAN [13], a parallelized version of DBSCAN, built from its respective source code repository and bound using Python's `ctypes`. The grid search itself has been parallelized with the `mpi4py` package, testing disjoint parameter combination sets. The underlying Message Passing Interface (MPI) implementation is an `OpenMPI 1.8.7` [11]. The script has been executed on a computer with a Fedora 22 operating system running the Linux kernel 4.10-200-fc22 and a four-core 64-bit Intel 2.4 GHz i7-4700MQ processor.

7 Conclusions

This work has studied the Koljö fjord observatory dataset collected over a timespan of a little over four years. The analysis goal has been to determine whether water mixing events, i.e., points in time where new water from the open sea or connected rivers flows into the fjord, can be automatically detected using a generic, data-driven analysis approach. A water mixing event is characterized by correlated peaks or drops of the considered time series variables water temperature, salinity and oxygen saturation.

From a data analysis point of view, the water mixing event detection is a binary outlier detection problem in a multivariate time series. This work extends univariate approaches based on clustering into a three-step multivariate approach, consisting of: 1. data smoothing to suppress noise, 2. outlier detection of the individual variables, and 3. a subsequent clustering of the results across all scales in order to identify the water mixing events. The effectiveness of the approach has been demonstrated using the Koljö fjord observatory data and ground truth data provided by the domain experts. An extensive grid search has yielded a production-ready model, that is able to identify water mixing events on test data with an F_1 measure of 0.8852, a *precision* of 0.931, and a *recall* of 0.8437. The found model parameters have sound values explainable by domain characteristics, as for example the smoothing filter size being an integral multiple of the tidal cycle.

One of the major advantages for the domain scientists is that water mixing events can now be tracked in an automated fashion without having to actively monitor the health of the fjord as new data comes in. This reduces operational costs, manual labor and allows a better time allocation in the project for domain-specific research. Moreover, the analysis pipeline could also be used as an automatic anti-fouling respectively reparation run reporting system. If the prediction of water

mixing events from multiple sensors in close proximity strongly differ, especially given there is only one or a small number of contrary results, this might be an indicator that these particular sensors need to cleaned or exchanged. As a result, costly ship rents for sensor reparation routines can be replaced by an on-demand schedule.

Due to the openness of the data and the analysis script, the experiment cannot only be reproduced, but also obtained and re-applied to similar problems. The data analysis algorithm for example is general enough, to be tailored to other water mixing event detection problems outside of the Koljö fjord observatory experiment, for example contamination monitoring in public water supply. For this, the analysis process and script can be directly used without essential changes other than the domain-specific data and ground-truth labels. Due to the chosen algorithms with low time and space complexity, the analysis pipeline can also scale to large amounts of data items.

One of the lessons learned of this study is that the simplified representation of each event as a central point only is challenging in practice due to the events' fuzzy nature, as defined by the domain experts. Future research should consider the automatic water mixing event detection as a traditional classification problem instead. For this, one would need precise labels for each of the data points, stating whether they mark the presence or absence of an event. Given that these labels exist, it would be possible to train classification models, such as support vector machines or neural networks, to predict for each individual pointer whether it is part of a water mixing event or not. A comparison of the obtained results with the method proposed in this study is the next logical step.

Conflict of Interest Statement

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Aanderaa Data Instruments AS (2016) Aanderaa Recording Doppler Current Meter 600. URL:http://www.aanderaa.com/media/pdfs/RDCP-600.pdf/, [Online; Accessed October, 07th 2016; 10:23 CEST]
2. Aanderaa Data Instruments AS (2016) Aanderaa Seaguard II DCP Doppler Current Profiler. URL:http://www.aanderaa.com/media/pdfs/seaguardii-dcp.pdf/, [Online; Accessed October, 07th 2016; 10:33 CEST]
3. Aanderaa Data Instruments AS (2016) Aanderaa Seaguard String System. URL:http://www.aanderaa.com/media/pdfs/seaguard-string-system.pdf/, [Online; Accessed October, 07th 2016; 10:34 CEST]
4. Andersson L, Rydberg L (1988) Trends in nutrient and oxygen conditions within the Kattegat: Effects of local nutrient supply. Estuarine, Coastal and Shelf Science 26(5):559–579
5. Arce G, McLoughlin M (1987) Theoretical analysis of the max/median filter. IEEE transactions on acoustics, speech, and signal processing 35(1):60–69
6. Atamanchuk D, Tengberg A, Aleynik D, Fietzek P, Shitashima K, Lichtschlag A, Hall PO, Stahl H (2015) Detection of CO2 leakage from a simulated sub-seabed storage site using three different types of CO2 sensors. International Journal of Greenhouse Gas Control 38:121–134
7. Bagnall AJ, Janacek GJ (2004) Clustering time series from ARMA models with clipped data. In: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 49–58
8. Cortes C, Vapnik V (1995) Support-vector networks. Machine learning 20(3):273–297
9. Diepenbroek M, Grobe H, Reinke M, Schindler U, Schlitzer R, Sieger R, Wefer G (2002) PANGAEA—an information system for environmental sciences. Computers & Geosciences 28(10):1201–1210
10. Ester M, Kriegel HP, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Knowledge Disocvery and Data Mining, vol 96, pp 226–231
11. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, et al (2004) Open mpi: Goals, concept, and design of a next generation mpi implementation. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, pp 97–104
12. Gariel M, Srivastava AN, Feron E (2011) Trajectory clustering and an application to airspace monitoring. IEEE Transactions on Intelligent Transportation Systems 12(4):1511–1524
13. Götz M, Bodenstein C, Riedel M (2015) HPDBSCAN: Highly Parallel DBSCAN. In: Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, ACM, p 2
14. Goutte C, Toft P, Rostrup E, Nielsen FÅ, Hansen LK (1999) On Clustering fMRI Time Series. NeuroImage 9(3):298–310
15. Götz M, Kononets M (2016) Auxiliary Material for the Koljöfjord Observatory Water Mixing Event Detection using DBSCAN. URL:http://hdl.handle.net/11304/8e3d1c07-96b6-4ab7-b4aa-f273ac8cbf74/, [Online; Accessed November, 17th 2016; 16:07 CET]
16. Han J, Kamber M, Pei J (2011) Data Mining: concepts and techniques - 3rd ed. Morgan Kaufmann
17. Hansson D, Stigebrandt A, Liljebladh B (2013) Modelling the Orust fjord system on the Swedish west coast. Journal of Marine Systems 113:29–41
18. Himberg J, Hyvärinen A, Esposito F (2004) Validating the independent components of neuroimaging time series via clustering and visualization. Neuroimage 22(3):1214–1222
19. Hodge VJ, Austin J (2004) A survey of outlier detection methodologies. Artificial intelligence review 22(2):85–126
20. Jiang D, Pei J, Zhang A (2003) Dhc: a density-based hierarchical clustering method for time series gene expression data. In: Bioinformatics and Bioengineering, 2003. Proceedings. Third IEEE Symposium on, IEEE, pp 393–400
21. Johnston F, Boyland J, Meadows M, Shale E (1999) Some properties of a simple moving average when applied to forecasting a time series. Journal of the Operational Research Society 50(12):1267–1271
22. Klise KA, McKenna SA (2006) Water quality change detection: multivariate algorithms. In: Defense and Security Symposium, International Society for Optics and Photonics, pp

62,030J–62,030J
23. Koljöfjord Observatory (2016) Koljöfjord Observatory Data. URL:http://koljofjord.cmb.gu.se/data/, [Online; Accessed June, 19th 2016; 15:07 CEST]
24. Koljöfjord Observatory (2016) PANGAEA Data Repository, Koljöfjord entries. URL:https://pangaea.de/search?q=KOLJOEFJORD, [Online; Accessed June, 19th 2016; 15:08 CEST]
25. Kononets M, Götz M (2016) Koljöfjord Observatory Preprocessed Data And Water Mixing Events. URL:http://hdl.handle.net/11304/f76da1d9-c61e-4250-beca-94d1b2803e77/, [Online; Accessed October, 07th 2016; 10:15 CEST]
26. Kut A, Birant D (2006) Spatio-temporal outlier detection in large databases. CIT Journal of computing and information technology 14(4):291–297
27. Liao TW (2005) Clustering of time series data—a survey. Pattern recognition 38(11):1857–1874
28. MacQueen J (1967) Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Oakland, CA, USA, 14, pp 281–297
29. Madsen H (2007) Time series analysis. CRC Press
30. Markus Götz (2016) PANGAEA Github Repository. URL:https://github.com/Markus-Goetz/pangaea, [Online; Accessed January, 13th 2016; 14:04 CET]
31. McKinney W (2012) Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. " O'Reilly Media, Inc."
32. Murray R, Haxton T, McKenna S, Hart D, Klise K, Koch M, Vugrin E, Martin S, Wilson M, Cruze V, et al (2010) Water quality event detection systems for drinking water contamination warning systems—development, testing, and application of canary. EPAI600IR-lOI036, US
33. Nordberg K, Filipsson HL, Gustafsson M, Harland R, Roos P (2001) Climate, hydrographic variations and marine benthic hypoxia in Koljö Fjord, Sweden. Journal of Sea Research 46(3):187–200
34. Pavlidis NG, Tasoulis DK, Plagianakos VP, Vrahatis MN (2006) Computational intelligence methods for financial time series modeling. International Journal of Bifurcation and Chaos 16(07):2053–2062
35. Perelman L, Arad J, Housh M, Ostfeld A (2012) Event detection in water distribution systems from multivariate water quality time series. Environmental science & technology 46(15):8212–8219
36. Powers D (2007) Evaluation: From Precision, Recall and F Factor to ROC, Informedness, Markedness & Correaltion. Sch Informatics Eng Flinders
37. Swedish Meteorological and Hydrological Institute (2016) Marina miljöövervakningsdata. URL:http://www.smhi.se/klimatdata/oceanografi/havsmiljodata/marina-miljoovervakningsdata, [Online; Accessed September, 19th 2016; 16:29 CEST]
38. University of Gothenburg (2016) Sven Lovén centrum för marin infrastruktur - Väderstation Kristineberg. URL:http://www.weather.loven.gu.se/kristineberg/, [Online; Accessed September, 19th 2016; 16:55 CEST]
39. Whitle P (1951) Hypothesis testing in time series analysis, vol 4. Almqvist & Wiksells
40. Zhao H, Hou D, Huang P, Zhang G (2014) Water quality event detection in drinking water network. Water, Air, & Soil Pollution 225(11):1–15

Appendix E

Paper V

Parallel Computation of Component Trees on Distributed Memory Machines

Markus Götz, Gabriele Cavallaro, *Member, IEEE*, Thierry Géraud, *Member, IEEE*, Matthias Book, and Morris Riedel, *Member, IEEE*

Abstract—Component trees are region-based representations that encode the inclusion relationship of the threshold sets of an image. These representations are one of the most promising strategies for the analysis and the interpretation of spatial information of complex scenes, since they allow efficient implementations of connected filters in a simple way. According to the sensor resolution and the application, images can represent vast and complex scenes by using large range of potential light intensity values. Existing sequential and parallel algorithms can not cope to this variety of data and present scalability limits in terms of memory and time. This work presents a novel distributed memory parallel algorithm for computing component trees.

Index Terms—Component Trees, Threshold Decomposition, Connected Component Labeling, High Performance Computing, MPI, Distributed-Shared-Memory-Hybrid.

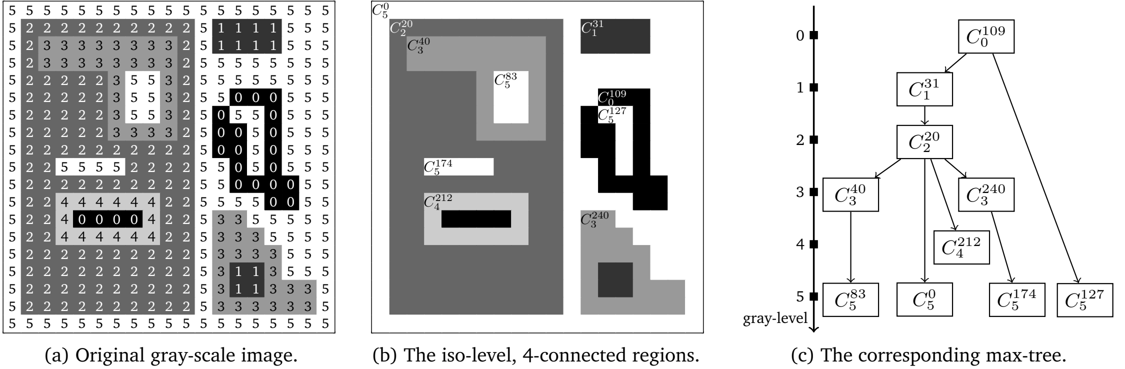
1 Introduction

Since the 1960s, mathematical morphology [1], [2] has become increasingly popular in the image processing community mainly due to its proven utility and rigorous mathematical description. The mathematical morphology framework provides a set of powerful operators for analyzing the spatial domain of images at the region-level—i.e., connected components—based on tree representations, called thresholds decompositions [3], [4]. These are based on tree representations of images which can be divided into two main groups [5]: hierarchies of segmentation—i.e., hierarchy of image partitions such as minimum spanning trees [6], alpha-trees [7], binary partition trees [8]—and threshold decompositions—i.e., hierarchy of regions such as component trees [4], [9], tree of shapes (ToS) [10] and multivariate tree of shapes [11]). Generally, tree structures are often considered richer in descriptive ability since they can be exploited for breaking down images into their fundamental elements which are easier to interpret with regards to the pixels. Component trees [4], [9], are thresholds decompositions that represent connected components [12] at every threshold level of an image in a hierarchical fashion, through parent relationships between nodes. The connected components organized in such trees can be filtered with different strategies [3], [4] and can model various types of connectivity [13].

Component trees have been popularized by connected operators, such as attribute filters [2], [3], which have been extensively used for the modeling of spatial information in images from remote sensing [14], [15], astronomy [16], [17] and medical scanning [18], [19]. Attribute filters are edge-preserving and flexible operators due to the preservation of contours in the processed objects and rely on multiple spatial measures or attributes. The possibility to perform a multi-attribute analysis, like attribute filters built by employing different attributes, enriches the extraction of spatial arrangement and improves the discrimination between different structures. In the presence of scenes with high complexity and heterogeneity, e.g., densely populated urban area, a complete modeling of the spatial information can be achieved through a multi-level analysis. It implies the decomposition of the original gray-level image obtained by applying a sequence of attribute filters according to a set of filter thresholds [20]. The result of this operation are the so-called attribute profiles [14]. They have been exploited mainly in remote sensing, e.g., classification [21], [22], data fusion [23] and change detection [24], as well as in medical imaging processing for the segmentation of computed tomographic images [25].

Nowadays, image processing applications can rely very high resolution data due to the continuing technological improvements of the sensor instruments. For example earth observation platforms have led to the increasing volume, acquisition speed and variety of sensed images, e.g., the World-View-3 satellite sensor (spatial resolution of 0.31m), or the AISA Dual airborne sensor (500 bands with spectral resolution of 2.9nm). The performances of traditional serial and parallel algorithms for computing the component trees are strictly correlated to the size and the quantization of the data. The size of remote sensed images is usually in the order of several gigabytes due to the depiction of vast and complex scenes. Consequently, they can not be stored or processed

- *M. Götz is with the Jülich Supercomputing Center, Wilhelm-Johnen-Straße 52428 Jülich, Germany, and the University of Iceland, 107 Reykjavik, Iceland.*
  E-mail: m.goetz@fz-juelich.de
- *G. Cavallaro is with the Jülich Supercomputing Center, Wilhelm-Johnen-Straße 52428 Jülich, Germany.*
  E-mail: g.cavallaro@fz-juelich.de
- *T. Géraud is with the EPITA Research and Development Laboratory (LRDE), Le kremlin-Bicêtre, France.*
  E-mail: theo@lrde.epita.fr
- *M. Book is with the University of Iceland, 107 Reykjavik, Iceland.*
  E-mail: book@hi.is
- *M. Riedel is with the Jülich Supercomputing Center, Wilhelm-Johnen-Straße 52428 Jülich, Germany, and the University of Iceland, 107 Reykjavik, Iceland.*
  E-mail: m.riedel@fz-juelich.de



(a) Original gray-scale image. (b) The iso-level, 4-connected regions. (c) The corresponding max-tree.

Fig. 1: Example of max-tree representation based on exemplary image and its components C_i^c, with the subscript c being the gray-level and the superscript i the canonical point uniquely identifying the component.

by algorithms designed for a single shared-memory machine. Furthermore, due to the high sensitivity of the new sensors, e.g., radiometric resolution, these images are characterized by an ample domain of integers or floating point values, which directly affect the processing time.

In this paper presents a shared- and distributed-memory hybrid algorithm for efficiently computing exact max-trees of integral gray-scale images as well as floating-point images. For this, the problem, i.e., the image is subdivided into equal-sized chunks that get assigned to all available distributing computing nodes. Then, each of the nodes computes a local, partial max-tree of the assigned chunk. A modified version of the shared-memory parallelized, depth-first, flooding max-tree algorithm proposed by Ouzounis *et al.* [26] is employed. Finally, the partial max-trees need to be merged into a correct, monolithic global representation. This is achieved by obtaining the iso-level edges of the boundary max-trees of the image chunks and iteratively resolving these per-gray-level, marking the major algorithmic challenge. In the proposed approach, the level connections are expressed as tuples—a data structural design that has been used by Flick el al. [27] for the distributed resolution of genomic graphs. This is a novelty in the mathematical morphology framework and the distributed computation of max trees.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to component trees. The subsequent Section 3 presents an overview over existing algorithms proposed in the literature. In Section 4 the proposed distributed, parallel max-tree algorithm is laid out. Complexity considerations and implementation details are explained in Section 5. A study of the algorithms strong and weak scaling as well as comparative study to the current state-of-the-art algorithm is presented in the experimental evaluation in Section 6. Finally, Section 7 concludes the paper, discussing the findings of this work and presents opportunities for future work.

2 Component Trees

Component trees were introduced by Jones [9], [28] as efficient image representations that enable the computation of advanced morphological filters in a simple way. These trees are actually hierarchical structures that encode the threshold sets and their inclusion relationship; in addition they allow efficient implementations of connected filters.

More formally, let $f : \Omega \rightarrow E$ be a discrete two-dimensional grayscale image, defined on a spatial domain $\Omega \subseteq \mathbb{Z}^2$ and taking values on a set of scalar values $E \subseteq \mathbb{Z}$. For any $\lambda \in \mathbb{Z}$, a lower $\mathcal{L}(f)$ and upper $\mathcal{U}(f)$ threshold set is defined by:

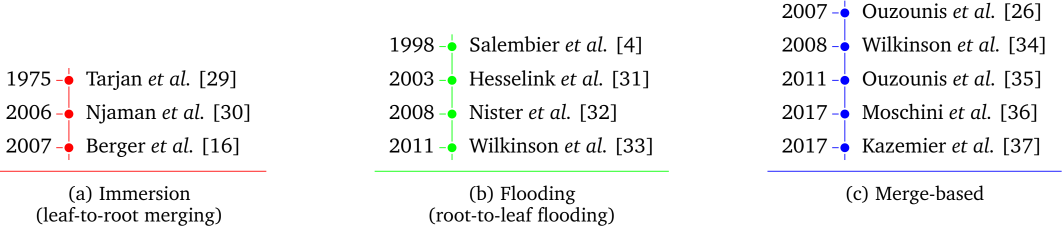
$$\mathcal{L}(f) = \{x \in \Omega, f(x) < \lambda\}, \quad (1)$$

$$\mathcal{U}(f) = \{x \in \Omega, f(x) > \lambda\}, \quad (2)$$

Let $\mathbb{P}(\Omega)$ be the power set of all the possible subsets of Ω. Given $X \in \Omega$, the set of connected components of X is denoted as $\mathcal{C}(X) \in \mathbb{P}(\Omega)$. If $\leq$ is a total relation, any two connected components $X, Y \in \mathcal{C}(\mathcal{L}(f))$ are either disjointed or nested. The min-tree and max-tree structures represent the components in $\mathcal{L}(f)$ and $\mathcal{U}(f)$ respectively with their inclusion relations. For example, Fig. 1c shows the max-tree structure of the image in Fig. 1a. The arrows denote the parent relation between the nested connected components that are identified in Fig. 1b. This is a simplified case that it is used for clarification purposes. In synthetic images that include more complex shapes (see the example in Section 4) or real scenaries the max-tree structure is less intuitive since its hierarchy is not driven by the inclusion relationship of connected components as it appears in Fig. 1.

3 Related Work

The selection of the most appropriate algorithm for computing the component trees shall be made according to the properties of the input image (i.e., size and pixel value quantization) and the processing resources available such as memory capacity and number of computing cores. Carlinet *et al.* [38] presented a comparative review of the state-of-the-art algorithms and provided detailed guidelines for selecting the most suitable algorithm. The algorithms are grouped into three main classes: immersion, flooding and merge-based. The algorithms that belong to the class immersion and flooding are also referred as leaf-to-root merging and root-to-leaf flooding methods, respectively [36]. Since this section is



(a) Immersion (leaf-to-root merging)   (b) Flooding (root-to-leaf flooding)   (c) Merge-based

Fig. 2: The main three classes of algorithms and their chronology.

not intended to repeat the review, Fig. 2 shows a timeline for each class of the algorithms that have been developed in the past and recent years.

Algorithm 1 Non-recursive algorithm [38].

1: **procedure** ProcessStack(r, q)
2: $\lambda \leftarrow f(q)$
3: Pop$(levroot)$
4: **while** $levroot$ **not** empty **and** $\lambda < f($Top$(levroot))$ **do**
5: INSERT_FRONT(S, r)
6: $r \leftarrow parent(r) \leftarrow$ Pop$(levroot)$
7: **if** $levroot$ empty **or** $f($Top$(levroot)) \neq \lambda$ **then**
8: Push$(levroot, q)$
9: $parent(r) \leftarrow$ Top$(levroot)$
10: INSERT_FRONT(S, r)

11: **function** Max-Tree(f)
12: **for all** p **do** $parents(p) \leftarrow -1$
13: $start_pixel \leftarrow$ any point in Ω
14: Push$(pqueue, start_pixel)$
15: Push$(levroot, start_pixel)$
16: $parent(start_pixel) \leftarrow$ INQUEUE
17: **loop**
18: `flood`:
19: $p \leftarrow$ Top$(pqueue)$; $r \leftarrow$ Top$(levroot)$
20: **for all** $n \in \mathcal{N}(p)$ such that $parent(p) = -1$ **do**
21: Push$(pqueue, n)$
22: $parent(n) \leftarrow$ INQUEUE
23: **if** $f(p) < f(n)$ **then**
24: Push$(levroot, n)$
25: **goto** `flood`
▷ p is done
26: Pop$(pqueue)$
27: $parent(p) \leftarrow r$
28: **if** $p \neq r$ **then** INSERT_FRONT(S, p)
29: **while** $pqueue$ **not** empty **do**
30: ▷ all points at current level done?
31: $q \leftarrow$ Top$(pqueue)$
32: ▷ Attach r to its parent
33: **if** $f(q) \neq f(r)$ **then** ProcessStack(r, q)
34: $root \leftarrow$ Pop$(levroot)$
35: INSERT_FRONT$(S, root)$

3.1 Flooding Algorithms

As mentioned in Section 1, since the proposed shared- and distributed-memory hybrid algorithm is based on flooding and merge-based strategies, the reader should refer to Tarjan [29], Najman *et al.* [30], Berger *et al.* [16] and Carlinet *et al.* [38] for a detailed explanation of the immersion algorithms.

The first flooding algorithm was proposed by Salembier *et al.* [4]. It is an efficient algorithm which retrieves the pixel at the lowest gray-level, i.e., root, through a scanning step and then it performs a propagation by flooding the neighbor at the highest level, i.e., a depth-first traversal of the connected components at higher intensities. Pixels in the propagation front are stored in a hierarchical queue composed by as many First In First Out (FIFO) queues as the number of gray-levels. It allows to directly access any pixel in the FIFO queue at a given level. Salembier's *et al.* [4] algorithm was rewritten in a non-recursive implementation by Hesselink *et al.* [31], later also by Nister *et al.* [32] and Wilkinson et al [33]. The algorithm presented by Wilkinson aims at solving the limitation of Salembier, the linear scaling with the number of gray-levels. Wilkinson has proposed to use a priority queue and a stack, a combination of the algorithms of Salembier *et al.* and Hesselink *et al.*, instead of using only a hierarchical queue for handling the pixel values during the flooding.

Carlinet *et al.* [38] have proposed a non recursive flooding algorithm variant of Salembier *et al.*, which has strong similarities with Wilkinson *et al.* and Nister *et al.* Due to the fact that the algorithm proposed in this work is based on it, the pseudocode is shown in Algorithm 3.1. The algorithm is divided into three stages: initialization, flooding and root fixing. In the initialization phase, a random point $start_pixel$ is chosen as the flooding point. This pixel is now considered as a canonical element, i.e. the representative of the connected component, and it is pushed on the stack $levroot$. The main purpose of the flooding phase is to compare the gray-level of each pixel p with its neighboring pixels n and enqueue those that have not yet been seen. The first processed pixels are p and the canonical element r of its component. These have the highest priority in the queue, i.e., the highest gray-level, and are on top of $levroot$ (p is not removed from the queue). The neighboring pixels n are pushed on the stack only if $f(n) > f(p)$, which immediately triggers the recursive call of *flood*. At a certain point, all the neighboring pixels of p will be either in the queue or already processed, meaning that the analysis of p has terminated. Following this, p is removed from the queue, $parent(p)$ is set to r, i.e., the canonical element. In order to ensure that the canonical element will be the last one inserted, p is added to S when $r = p$. After p is removed from the queue, the canonical element r is attached to its parent only when the level component has been fully

processed. The *ProcessStack* procedure is called when q has a different level than p. It pops the stack, it sets the parent relationship between the canonical elements and it inserts them in S until the top component has a level no greater than $f(q)$. When the stack gets empty or the top level is lower than $f(q)$, then q is pushed on the stack as the canonical element of a new component. The algorithm ends when all points in queue have been processed, then S only misses the root of the tree which is the single element that remains on the stack.

3.2 Merge-based algorithms

The natural way to implement a parallel algorithm is to divide the original image domain and compute the max-tree on each sub-image using any algorithm by Wilkinson *et al.* [34], Ouzounis *et al.* [26] or Matas *et al.* [39] from Fig. 2. In order to compute this partition, the image should be split in Np connected disjoint regions, which is the union that forms the entire image domain. During this step, the image is split into a reasonable number of chunks, which reflects the underlining processing architecture, e.g., number of threads available. For instance, when the number of image chunks is lower than the number of threads the domain is not decomposed enough and the distribution of the computations is not yet optimal (load imbalance). Some threads will idle while having to wait for other threads to finish. Once all subtrees are generated, they can be merged into a single global tree. This is a non-trivial phase as it requires that the gray-levels of the connected components are merged and their parent relationships updated. An efficient algorithm to merge the max-trees resulting from any arbitrary image sections can be found in Wilkinson et al [34].

3.3 Connected Component Labeling

The main problem of generating the max-tree can be split into two parts: find the connected components and establish their hierarchy. The task of grouping connected pixels within an image can be seen as the well-known Connected Component Labeling problem [40]. This is an important step for a large number of applications and it relies firstly on finding which parts of an object, e.g., binary images, gray-levels images, data with higher dimensionality, etc., that are physically connected, based on a connectivity rule, and secondly, to label them. Iverson *et al.* [41] provide an evaluation of connected-component labeling algorithms in the context of distributed computing, when data that need to be processed in a given application usually require large processing power and distributed-memory machines. They conclude that there is an unavoidable compromise to find between memory and processing time. Most of the available parallel algorithms are problematic especially in terms of memory requirements. For instance, the merging step could end up on a single node resulting in an unbalanced scenario. However, algorithms that try to solve this problem provide poor scaling results in terms of processing time. At the same time Flick el al. [27] proposed a scalable distributed-memory algorithm to overcome this problems raised by Iverson. They have aimed to solve issues such as excessive memory usage, extra computation and communication of the processors, and load balancing. The main idea of the algorithm is similar to the Shiloach-Vishkin algorithm [42] in that it transforms the problem into finding weakly connected components within the Bruijn directed graph [43]. It will be shown in Section 4 that the proposed algorithm uses the notion of tuples and inverse doubling in order to connect the overlap zones between the split regions and resolves the connected components and their corresponding parenthood.

4 Distributed Max-Tree Algorithm

4.1 Definitions and Notation

A two-dimensional gray-scale image f can be seen as an undirected graph $\mathcal{G} = (V, E)$. V represents a set of vertices—the pixels of the image—and $n = |V|$ the total number of pixels. Then E, a number of edges or non-ordered pairs of vertices (v_i, v_j), with $i, j \in [0, n[$, which model the neighborhood relationship of the pixels. Classically, images are either four- or eight-connected [44], meaning the top, left, right and bottom neighbors, respectively including the diagonals, are considered connected neighbors. The entire graph $\mathcal{G}$ is said to be connected if, for any $p, q \in V$, there exists a path from p to q, which is a sequence of $s > 1$ vertices—i.e., $p = p_1, ..., p_s = q$—such that every $p_i \in V$, and any two successive pixels of the sequence are adjacent $e_{p_i,p_{i+1}} \in E$. Given this definition a connected component $\mathcal{CC}$ is a subgraph of $\mathcal{G}$ if $V_{\mathcal{CC}} \subseteq V_{\mathcal{G}} \wedge E_{\mathcal{CC}} \subseteq E_{\mathcal{G}}$, $\forall i \in V_{\mathcal{CC}} : f(i) = c$ and $\mathcal{CC}$ is not connected to any other subgraph of $\mathcal{G}$. A connected component can be either weak or strong connected, depending on the path length s. Weak connected graphs can have an arbitrary path length, while for strong connected graphs $s = 2$ holds. Furthermore, if not stated otherwise, the following symbols are defined for the remainder of the document: h and w is the height and the width of the image f,respectively. The entire image has a gray-level depth d, i.e., the number of different grayvalues c. Concerning parallelization, the number of available distributed compute nodes is p, while the local number of shared-memory threads is labeled with t. For the explanation of the distributed resolution, it is also necessary to introduce what is coined a tuples. These are essentially quartuples, mathematical tuples with four components, of the form $\langle c_i, p_k, c_j, p_l \rangle$ with c_i and c_j being two gray-values and p_k and p_l two vertices. They are used to explicitly express edges $e \in E$ of the image f of a canonical point with a certain gray-value to a different canonical point of a given other gray-value.

4.2 Concept

The general nature of the distributed max-tree algorithm can be described as divide-and-conquer. This means, that the entire problem, i.e., the image, is divided into sub-images for which the respective max-tree is computed and that are then successively merged along the division boundaries. The major algorithmic challenge lies in the latter stage. It requires to solve two demanding graph theory sub-problems—connected component labeling and graph canonicalization—in distributed memory environments. This work proposes an iterative, parallel merging algorithm based on explicit expression of the max-tree edges as directed tuples. For this, each sub-image overlaps the respective next by a one-pixel high halo zone for which all of the tuples of the boundary

max-trees are generated. Conflicts within the tuples are resolved by searching for the most optimal tuple candidates, i.e., the smallest canonical point for the connected labeling or the next closest root for the graph canonicalization, and remapping the remaining tuples accordingly. After the tuples are resolved, they are send back to the respective sub-images of origin, locally applied and thus create the globally create max-tree. Algorithm 2 sketches the proposed strategy.

Algorithm 2 Pseudocode of the proposed distributed max-tree algorithm.

```
 1: @parallel
 2: function DISTRIBUTED-MAX-TREE(f)
 3:     p ← number of nodes
 4:     r ← processor id in range [0, p[
 5:     t ← number of threads
 6:     f' ← LOAD-PARTIAL-IMAGE(f, r, p)
 7:
 8:     parents' ← LOCAL-MAX-TREE(f', t)
 9:
10:     area_tuples ← CONNECT-HALOS(f', parents')
11:     root_tuples ← HALO-ROOTS(f', parents')
12:     tuples ← RESOLVE(area_tuples, root_tuples)
13:
14:     tuples' ← REDISTRIBUTE(tuples)
15:     parents ← APPLY(tuples', parents')
16:
17:     return parents
```

4.3 Local Max-Tree Algorithm

For the local computation one can in principle employ any correct max-tree algorithm. The proposed solution specifically utilizes a modified version of Salembier's depth-first, flooding algorithm. The major algorithm structure is reformulated to be non-recursive, as suggested by Carlinet *et al.* [38] for example, but additionally enhanced to always use the minimal pixel index as canonical point for an iso-level and to yield better computational performance. Algorithm 3 presents the corresponding pseudocode.

The first change can best be seen in line 26. In contrast to the original non-recursive variant, the canonical area point is not chosen at the beginning of an iso-level processing—as it may not yet be the canonical minimum—but rather constantly maintained throughout the process. This is done by keeping the current area minimum at a specific place, e.g., the front of the pixel vector, and compared to on insertion of new elements. Only after all pixels of the entire iso-level is found, the canonical point is assigned in the parent image, see also line 31, and thus minimality of the index guaranteed.

Moreover, when one considers the computational performance of the algorithm, the proposed modifications also allows for faster computation. Before, each gray-level had its one hierarchical queue in Salembier's original algorithm formulation or a singular in Carlinet's non-recursive reformulation. This approach scales logarithmically with both, the number of gray-levels as well as the number of pixels per channel. In the proposed variant the gray-levels are keys to a map, called *stacks*, that has vectors for the corresponding pixels as keys. Then, insertions only scale logarithmically with

Algorithm 3 Pseudocode of the modified version of Salembier's depth-first, flooding-based max-tree algorithm.

```
 1: function MAX-TREE(f)
 2:     stacks ← {}                                   ▷ Initialization
 3:     pixels ← {}
 4:     children ← [ ]
 5:     for all p ∈ f do
 6:         parents(p) ← −1
 7:         deja_vu(p) ← false
 8:
 9:     start_pixel ← any point in Ω                  ▷ Seed pixel
10:     start_grayv ← f(start_pixel)
11:     deja_vu(start_pixel) ← true
12:     PUSH(stacks(start_grayv), start_pixel)
13:     PUSH(pixels(start_grayv), start_pixel)
14:
15:     while not EMPTY(stacks) do                    ▷ Depth-first
16:         flood:
17:         grayv ← MAX-KEY(stacks)
18:         pixel ← POP(stacks(grayv))
19:         for all n ∈ N(p) do
20:             if deja_vu(n) then continue
21:             deja_vu(n) ← true
22:             PUSH(stacks(f(n), n))
23:             PUSH(pixels(f(n), n))
24:             if TOP(stacks) > BACK(stack) then
25:                 ▷ Ensure canonical point is in front
26:                 SWAP(TOP(stacks), BACK(stacks))
27:             if grayv < f(neighbor) then
28:                 PUSH(stacks(grayv), pixel)
29:                 goto flood
30:
31:         if EMPTY(stacks(grayv)) then              ▷ Iso-level done
32:             c ← pixels(grayv)
33:             for all p ∈ pixels(grayv) do parents(p) ← c
34:             ERASE(pixels(grayv))
35:             ERASE(stacks(grayv))
36:
37:             if EMPTY(stacks) then                 ▷ Attach children
38:                 merge ← MAX-KEY(stacks)
39:             else
40:                 merge ← grayv
41:             while not EMPTY(children) and
                    ↪ BACK(children).grayv > merge do
42:                 child ← POP(children)
43:                 parents(p) ← child.pixel
44:             PUSH(children, ⟨merge, c⟩)
45:
46:     if not EMPTY(children) then                   ▷ Attach root children
47:         root ← BACK(children).grayv
48:         for all c ∈ children do parents(c) ← root
49:
50:     return parents
```

0 (00)	2 (01)	0 (02)	0 (03)	0 (04)	0 (05)	0 (06)
0 (07)	3 (08)	3 (09)	0 (10)	0 (11)	4 (12)	0 (13)
0 (14)	0 (15)	4 (16)	0 (17)	0 (18)	4 (19)	0 (20)
2 (21)	0 (22)	4 (23)	0 (24)	0 (25)	4 (26)	0 (27)

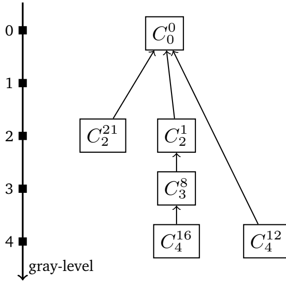
(a) Partial image of p_0

0 (00)	2 (01)	0 (02)	0 (03)	0 (04)	0 (05)	0 (06)
0 (07)	3 (08)	3 (09)	0 (10)	0 (11)	4 (12)	0 (13)
0 (14)	0 (15)	4 (16)	0 (17)	0 (18)	4 (19)	0 (20)
2 (21)	0 (22)	4 (23)	0 (24)	0 (25)	4 (26)	0 (27)
0 (28)	0 (29)	4 (30)	0 (31)	0 (32)	4 (33)	0 (34)
0 (35)	0 (36)	4 (37)	4 (38)	4 (39)	4 (40)	0 (41)
0 (42)	0 (43)	2 (44)	0 (45)	1 (46)	2 (47)	0 (48)

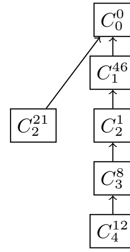
(b) Unpartitioned image

2 (21)	0 (22)	4 (23)	0 (24)	0 (25)	4 (26)	0 (27)
0 (28)	0 (29)	4 (30)	0 (31)	0 (32)	4 (33)	0 (34)
0 (35)	0 (36)	4 (37)	4 (38)	4 (39)	4 (40)	0 (41)
0 (42)	0 (43)	2 (44)	0 (45)	1 (46)	2 (47)	0 (48)

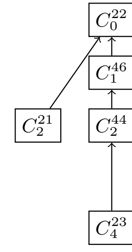
(c) Partial image of p_1



(d) Max-tree of p_0's partial image.



(e) Unpartitioned image



(f) Max-tree of p_1's partial image.

p_0		p_1
$\{4:[\langle 4,16,4,12\rangle,\langle 4,12,4,16\rangle]\}$	**Connect halos**	$\{0:[\langle 0,22,0,0\rangle,\langle 0,0,0,22\rangle]$ $4:[\langle 4,23,4,12\rangle,\langle 4,12,4,23\rangle]\}$
$\{4:[\langle 4,12,0,0\rangle,\langle 4,16,3,8\rangle]$ $3:[\langle 3,8,2,1\rangle]$ $2:[\langle 2,1,0,0\rangle]\}$	**Halo roots**	$\{4:[\langle 4,23,2,44\rangle]$ $2:[\langle 2,21,0,0\rangle,\langle 2,44,1,46\rangle]$ $1:[\langle 1,46,0,22\rangle]\}$
	Resolve	
	C_4	
$[\langle 4,12,4,16\rangle,\langle 4,12,4,23\rangle]$	RESOLVE-AREA	$[\langle 4,16,4,12\rangle,\langle 4,23,4,12\rangle]$
$[\langle 4,12,0,0\rangle,\langle 4,16,3,8\rangle,\langle 4,16,4,12\rangle]$	*Normalize root tuples*	$[\langle 4,23,2,44\rangle,\langle 4,23,4,12\rangle]$
$[\langle 4,12,0,0\rangle,\langle \mathbf{4},\mathbf{12},\mathbf{2},\mathbf{44}\rangle]$	RESOLVE-ROOTS	$[\langle \mathbf{4},\mathbf{12},\mathbf{3},\mathbf{8}\rangle]$
$[\langle \mathbf{3},\mathbf{8},\mathbf{0},\mathbf{0}\rangle,\langle \mathbf{3},\mathbf{8},\mathbf{2},\mathbf{44}\rangle]$	*Resolved tuples*	$[\langle 3,8,4,12\rangle]$
	C_3	
$[\langle 3,8,0,0\rangle,\langle 3,8,2,1\rangle]$	RESOLVE-ROOTS	$[\langle 3,8,2,44\rangle,\langle 3,8,4,12\rangle]$
$[\langle \mathbf{2},\mathbf{1},\mathbf{0},\mathbf{0}\rangle,\langle 2,1,3,8\rangle]$	*Resolved tuples*	$[\langle \mathbf{2},\mathbf{44},\mathbf{2},\mathbf{1}\rangle,\langle \mathbf{2},\mathbf{1},\mathbf{2},\mathbf{44}\rangle,\langle \mathit{4},\mathit{12},\mathit{3},\mathit{8}\rangle]$
	C_2	
$[\langle 2,1,2,44\rangle]$	RESOLVE-AREA	$[\langle 2,1,2,44\rangle]$
$[\langle 2,1,0,0\rangle,\langle 2,1,3,8\rangle,\langle 2,21,0,0\rangle]$	*Normalize root tuples*	$[\langle 2,44,1,46\rangle,\langle \mathit{2},\mathit{44},\mathit{2},\mathit{1}\rangle]$
$[\langle 2,1,0,0\rangle,\langle \mathbf{2},\mathbf{1},\mathbf{1},\mathbf{46}\rangle]$	RESOLVE-ROOTS	$[\langle 2,1,3,8\rangle,\langle 2,21,0,0\rangle]$
$[\langle \mathbf{1},\mathbf{46},\mathbf{0},\mathbf{0}\rangle,\langle 1,46,2,1\rangle]$	*Resolved tuples*	$[\langle \mathit{3},\mathit{8},\mathit{2},\mathit{1}\rangle,\langle 0,0,2,21\rangle]$
	C_1	
$[\langle 1,46,0,0\rangle,\langle 1,46,0,22\rangle]$	RESOLVE-ROOTS	$[\langle 1,46,2,1\rangle]$
$[\langle 0,0,1,46\rangle,\langle \mathbf{0},\mathbf{0},\mathbf{0},\mathbf{22}\rangle,\langle \mathbf{0},\mathbf{22},\mathbf{0},\mathbf{0}\rangle]$	*Resolved tuples*	$[\langle \mathit{2},\mathit{1},\mathit{1},\mathit{46}\rangle]$
	C_0	
$[\langle 0,0,0,22\rangle]$	RESOLVE-AREA *iteration 1*	$[\langle 0,22,0,0\rangle]$
$[\langle 0,0,1,46\rangle,\langle 0,0,2,21\rangle]$	*Normalize root tuples*	$[\langle 0,22,0,0\rangle]$
$[\langle \mathit{1},\mathit{46},\mathit{0},\mathit{0}\rangle,\langle 2,21,0,0\rangle]$	*Resolved tuples*	$[\langle \mathit{0},\mathit{22},\mathit{0},\mathit{0}\rangle]$

Fig. 3: Toy example demonstrating the tuple resolution of the distributed max-tree algorithm.

the number of gray-levels, for locating the respective vector in the map, but the actual push operation happens in constant time. Especially in images with a large amount of pixels, this can drastically reduce computation time.

For the shared-memory parallelization the strategy explained by Ouzounis *et al.* [26] has been chosen. The local image partition of the node distribution step, is again horizontally, virtually partitioned in t equally sized chunks, without overlap, and assigned to one of the t threads. For each of the partitions the partial max-tree is computed using the introduced algorithm. The virtual images boundaries are realized by excluding respective pixels in the neighborhood searches. Finally, the partial max-trees are merged using the `connect` function, equally presented by Ouzounis [26]. Minor changes have been made to ensure that the canonical points of each iso-level is guaranteed to be minimal. This can be achieved by performing a look-ahead on the upcoming elements of the merge stacks and potentially swap them if required. After the local computation, one would obtain the partial max-trees depicted in Fig. 3d and Fig. 3f.

4.4 Tuple Generation

The tuple generation of the partial images is subdivided into two major steps. First, there is the generation of area tuples—i.e., tuple that connect the divided iso-level—and, second, the root tuples are created that express the max-tree branches in the halo areas.

The former is achieved by performing two prefix sums across the canonical points in the halo zones of the partial images. In the first run, the canonical points are broadcast downwards across the partial images, while the second, reverse, prefix-sum back-propagates potential merges of areas. For the example in Fig. 3 this means that p_0 broadcasts the canonical-halo-vector $\langle 21, 0, 16, 0, 0, 12, 0\rangle$ to p_1, which are almost exclusively more optimal choices—i.e., smaller—for the canonical points of the iso-levels. The only exception to that is the iso-level with the gray-value 4. p_1 links the independent segments of p_0 with the canonical points 12 and 16. Therefore, p_1 needs to back-propagate the vector $\langle 21, 0, 12, 0, 0, 12, 0\rangle$. Ultimately, this allows both nodes to generate the following non-redundant, area tuples: $p_0 : \langle 4, 16, 4, 12\rangle$ and $p_1 : \langle 0, 22, 0, 0\rangle, \langle 4, 23, 4, 12\rangle$ including each inverse, which is required for faster tuple resolution and will be explained in Section 4.5. In scenarios with more processing nodes this non-commutative operation is reapplied to the merged partial image. This means for the given example that the canonized top halo of p_0 and the canonized bottom halo of p_1 are passed on. The entire prefix-sum operation can be efficiently implemented using a logarithmic merge tree across all available nodes.

The second set of tuples are generated by traversing the boundary max-trees, i.e., the ones in the halo zones, recursively upwards to the root. For each edge of the tree a corresponding tuple is generated with the canonical point of a child iso-level pointing to the canonical point of its parent. In the partial image of processor p_0, depicted in Fig. 3a, for example, the tuple $\langle 2, 21, 0, 0\rangle$ is generated for the boundary pixel with the index 21. All resulting tuples are shown in Fig. 3 in the **Halo roots**-step. In order to optimize the performance of the tuple generation already visited branches can be skipped, which will also avoid redundant tuples. The entire step can be computed fully local and does not require any data exchange with other cores.

4.5 Distributed Tuple Resolution

The resolution of the tuples happens essentially in the same mode as the generation of tuples. First, the area tuples are resolved, which requires the iterative resolution of the weakly connected area components into strong ones, and then, second, the resolution of the roots for the normalized areas. The respective high-level pseudocode is displayed in Algorithm 4.

Algorithm 4 Pseudocode of a single iterations of the distributed area tuple resolution.

```
 1: @parallel
 2: function RESOLVE(area, roots)
 3:     tuples ← []
 4:
 5:     loop
 6:         all_done ← ALLREDUCE(EMPTY(roots), And)
 7:         if all_done then break
 8:
 9:         grayv ← ALLREDUCE(MAX-KEY(roots), Max)
10:         unresolved ← true
11:         while unresolved do
12:             GLOBAL-SORT(area)
13:             rules ← RESOLVE-AREA(area(grayv))
14:             unresolved ← REMAP(area(grayv), rules)
15:             ALLREDUCE(unresolved, Or)
16:
17:         resolved ← RESOLVE-ROOTS(grayv, area, roots)
18:         tuples ← CONCAT(tuples, resolved)
19:
20:     return tuples
```

In general, the area tuple resolution is inspired by the distributed connected component labeling algorithm presented by Flick et al [27]. The goal is to turn a weakly connected graph, here the areas, into a strongly connected one, meaning directly pointing to the correct canonical point of an area without intermediate, transitive connections. This is achieved by following the graph edges until the most optimal, i.e., smallest pixel point is found. However, in distributed memory environments it might no be able to follow all paths directly as they might be residing on a different machine. To overcome this, the longest partial graph paths are computed and iteratively shortened until the strong graph is found. For this, all tuples are globally sorted, i.e., across all nodes, and locally linearly scanned for the most optimal candidate, remapped and saved. This process is repeated until convergence is achieved, which is equal to having no tuples remapped in the current iteration. Technically, there are two challenges involved in this.

First, there is the problem of globally sorting the tuples. This means that all tuples need to be partially ordered, so that the smallest element is on node with the smallest rank and the maximal tuple on the node with the highest rank. For this, the distributed max-tree algorithm uses an enhanced version

of the parallel sorting by regular sampling algorithm. In the variant that is proposed here, the number of tuples are in addition automatically balanced, in order to have keep the workload on each node equal.

Second, if the tuples are balanced, then the partial graphs of a component, i.e., all the tuples with the same origin, must not necessarily reside in the memory of the same node or might even span multiple nodes. A good example can be seen in Figure 3 of RESOLVE-AREA step of gray-channel C_2. Therefore, the distribute max-tree algorithm must connect these partial graphs in each iteration. This can be achieved by exchanging the ends of the sorted tuple chain including the found canonical point with the direct neighbors. Given that the neighboring node proposes a better canonical point, it is adopted instead of the one found locally. Transitivity is achieved by logarithmically merging these chains across all available machines using a prefix sum, first from left to right across the ranks, and then in reverse to propagate better options back.

Furthermore, each tuple exists, twice. Once in the "correct direction" pointing from the larger pixel index to the lower index and its inverse, pointing from low to high. Whenever a tuple is remapped, the tuple is flipped, i.e., the direction is changed in order to back propagate this change to its inverse. The reason behind this is, that the inverse tuple might have found an even more optimal canonical point coming from the other side of the chain, due to say half circular structures on the image. Both tuples are then updated and the transitive chain shortened much quicker. As an effect of this, the number of iterations heavily reduces, as introduced by Flick et al [27]. The pseudocode of the area tuple resolution can be found in Algorithm 5.

Algorithm 5 Pseudocode of the area remapping algorithm.

```
 1: function RESOLVE-AREA(area)
 2:     rules ← {}
 3:
 4:     for all tuple ∈ area do
 5:         from ← tuple.from
 6:         to ← tuples.to
 7:         if to > from then SWAP(to, from)
 8:         canonical ← CANONIZE(rules, from)
 9:         min ← MIN(canonical, to)
10:         max ← MAX(canonical, to)
11:         rules[from] = min
12:         rules[max] = min
13:
14:     ends ← [FRONT(area), BACK(area)]
15:     PREFIX-SUM(ends, rules, Min, Left)
16:     PREFIX-SUM(REVERSE(ends), rules, Min, Right)
17:
18:     return rules
```

After the canonicalization of the weak connected components into strong connected components, the roots for each of the canonical points must be found. This is done by merging the area tuples and root tuples, global sorting and linearly scanning for the most optimal root. Then, the tuples are scanned again and compared to the found most optimal root. There are four possible options how a tuple can relate to it, which are also shown in the pseudo code in Algorithm 6. First, the tuple origin gray-value is small, i.e., further up in the tree, than the root, then a tuple needs to be created that connects both of them transitively from the root to the tuple (see case 2). Second, the gray-value of the tuple and the root match, but the root has a smaller canonical point (see case 3) In this case, the root and its connected component as well as the one of the neighbor are iso-level connected via the area of the current tuple. This means according area remapping tuples need to be created, including their inverse and added to the area tuples. Third, the current tuple already points correctly to the root (see case 4), then the tuple is inverted, essentially pointing the "wrong" way around from the root to the lower area and pushed into the root's gray-valued tuple bucket. The reason behind this is, that the canonical point of the root might still be changed during the area resolution phase of its gray-value. Therefore, it may not be marked as finished yet, but kept until the respective gray-level of the roots has been resolved. Finally, the fourth condition (see case 1) is meant for inverted root tuples. After their normalization, they are flipped yet again into the "correct" order, i.e., pointing from high gray-values to low gray-values and send back to its respective tuple bucket.

4.6 Obtaining the Global Parent Image

After the tuples have been resolved the global parent image can be obtained by redistributing the tuples back to their original sub-image. For this, the each tuple is send back to the node, that contains the pixel with the index of the second tuple component, independent whether it is a area or root tuple. Each of the area tuples is then stored in an associative container, mapping from the original points to the canonical point. Subsequently, while iterating over the image, each of the pixels is normalized to its correct canonical points using said data structure. The root tuples are handled slightly differently. Each tuples is visited once and the pixel index at the from part of the tuples is simply set to the destination of the tuple. After this step, each node posses the correct partial parent image representing the global max-tree.

5 IMPLEMENTATION

The proposed parallel algorithm has been implemented in C++ and is available on the open-source, scientific code reproducibility platform CodeOcean [45]. The coarse-grained parallelization across multiple nodes has been realized using the Message Passing Interface (MPI) [46]. For the shared-memory implementation C++11 native threads have been used. The algorithm accepts data loaded from files in the Hierarchical Data Format 5 (HDF5) format [47], which it will also store the resulting parent image to.

5.1 Complexity

In this section the time and space complexity for the algorithm steps of the distributed max-tree computation are laid out. A summary can be found in Table 1. The used symbols are explained in Section 4.1. All formulas are given for the worst-case scenario.

The time and space complexity for loading the sub images can be straight-forward inferred and amount to the number

Algorithm 6 Pseudocode of the distributed resolution of the root tuples.

```
 1: function RESOLVE-ROOTS(grayv, area, roots)
 2:   combined ← CONCAT(area(grayv), roots(grayv))
 3:   GLOBAL-SORT(combined)
 4:
 5:   best_roots ← FIND-BEST-ROOTS(combined)
 6:   NORMALIZE(combined)
 7:   ends ← [FRONT(combined), BACK(combined)]
 8:   PREFIX-SUM(ends, Area-And-Root, Left)
 9:   PREFIX-SUM(ends, Area-And-Root, Right)
10:   GLOBAL-SORT(combined)
11:
12:   for all tuple ∈ combined do
13:     root ← best_root[tuple.from]
14:     if tuple.grayv > tuple.n_grayv then                ▷ Case 1
15:       PUSH(roots(tuple.grayv), INVERT(tuple))
16:     else if tuple.grayv < root.grayv then              ▷ Case 2
17:       PUSH(roots(root.grayv), ⟨root.grayv,
            ↪ root.from, tuple.n_grayv, tuple.to⟩)
18:     else if tuple.n_grayv = root.grayv and
          ↪ root.pixel < tuple.to then                     ▷ Case 3
19:       t_to ← CANONIZE(tuple.to)
20:       r_to ← CANONIZE(root.to)
21:       n_c ← tuple.n_grayv
22:       min_to ← MIN(t_to, r_to)
23:       max_to ← MAX(t_to, r_to)
24:       PUSH(area[n_c], ⟨n_c, tuple.to, n_c, min_to⟩)
25:       PUSH(area[n_c], ⟨n_c, min_to, n_c, tuple.to⟩)
26:       PUSH(area[n_c], ⟨n_c, max_to, n_c, min_to⟩)
27:       PUSH(area[n_c], ⟨n_c, min_to, n_c, max_to⟩)
28:     else                                               ▷ Case 4
29:       PUSH(roots(root.grayv), ⟨root.grayv,
            ↪ root.from, tuple.grayv, tuple.from⟩)
30:
31:   return tuples
```

TABLE 1: Overview of the worst-case time and space complexity of the distributed max-tree algorithm steps.

	Time	Space
Image chunking	$\mathcal{O}(\frac{n}{p})$	$\mathcal{O}(\frac{n}{p})$
Local max-tree	$\mathcal{O}(\frac{n}{p \times t} \times log(d) + w \times log(t))$	$\mathcal{O}(\frac{n}{p} + t \times w)$
Area tuple generation	$\mathcal{O}(w \times log(p))$	$\mathcal{O}(w)$
Root tuple generation	$\mathcal{O}(w \times d)$	$\mathcal{O}(w \times d)$
Tuple resolution	$\mathcal{O}(k \times d \times \frac{w}{d} \times log(w \times d)) \times log(p)$	$\mathcal{O}(w \times d)$
Redistribution	$\mathcal{O}(w \times d)$	$\mathcal{O}(w \times d)$
Application	$\mathcal{O}(w \times d + \frac{n}{p} \times log(w \times d)))$	$\mathcal{O}(w \times d)$

of total pixels divided by the amount of available processing nodes, as each of the receives an equally size chunk of the entire problem. For the local max-tree computation each of the t thread needs to allocated the part of parent image, that is equal in size to the processed raw image, plus and additional area remapping that is solely dependent on the image width, explaing the space complexity. The computational complexity consists of the linear image-scan for each thread and sub-image, which in turn need to do look-ups into the associative container for the *stacks*, resulting in the first summand. However, each thread needs to be merged with its direct neighbors, which can be done in logarithmic fashion as explained in Section 4.3, along the virtual split boundaries, i.e., the width of the image.

The next two steps involve the generation of the tuples. In the worst case, for each of the boundary pixels—again the width of the image—a tuple needs to created, resulting in according space and time complexity. Although, in practice the number will most of the times be much lower, because the boundary zones mostly consists of connected flat zones with shared parents nodes, resulting in early outs. In fact, the average complexity should therefore be closer to $\mathcal{O}(w * log(\frac{d}{2}))$, but is for obvious reasons dependent on the analyzed data.

Contrary to root tuples, which can be generated entirely local, area tuples needs to be stitched together across all processing nodes, resulting in the additional factor for the time complexity. The main resolution consists of k iterations of sorting, linearly scanning and remapping the tuples for each of the available gray-levels of the gray depth d. It is assumed here that the number of tuples per gray-level is more or less evenly distributed, resulting in the purposefully chosen term $\frac{w}{d}$ (it would actually cancel out with d). The sorting adds both logarithmic components, over the number of tuples and nodes, as it requires reordering them across all machines. One of the major uncertainty factors is the iteration constant k, which is dependent on the data. In the worst case, k is equal to the number of processing nodes p, given a single tuple needs to visit each single machine due to transitivity. However, in practice, the iteration count will remain low, typically only one or two, even for a high number of nodes, due to the area stitching step and tuple inversions, effectively minimizing the canonicalizations. Finally, the resolved tuples need to be send back to the partial image of origin and locally applied. The later step requires iterating over the whole image and normalizing each of the pixels using an associative data structure with logarithmic look up time. This is the reasons for the second summand in the time complexity equation. The first can be explained by the changing roots by directly assigning them while iterating through the tuples.

Generally, the algorithm has complexity classes that are either linear or linear-logarithmic, supporting good scalability overall. The only bottleneck seems to be the iterative constant k that could potentially degrade into the number of used compute nodes p. However, this is rarely the case in practical use, but is a good subject for further research.

5.2 Implementation Details

For the algorithm implementation the Message Passing Interface (MPI) [48] programming framework has been used. It provides low-level network communication primitives to exchange one-to-one and many-to-many messages between the participating compute nodes. Efficient algorithms usually rely on the later category of operations, the so-called *collectives*, due to possibility of achieving a logarithmic scaling over the number of cores. For this reason, the distributed max-tree algorithm employs collective `MPI_Scan` calls to generate the area-connecting tuples of the halo zones.

At first, a right oriented scan, that is from the first to the last image chunk, broadcasts the minimal canonical points to subsequent image partitions, while, in a second, left oriented

scan, potentially merged areas are reported back. Due to the fact, that the top and bottom halo of a image chunk may have entirely disjoint connected components per pixel, the worst case memory complexity of the exchanged buffer is $\mathcal{O}(p * w * 2)$. This is highly undesirable, as it scales both, with the number of processing nodes as well as the width of the image. One can realize this operation more efficient, if only the outer boundaries of the already merged images are communicated and the intermediary remapping rules are memorized in a data structure, e.g., a map, across the two `MPI_Scan` calls. Then, the memory complexity of the sent buffer simply becomes $\mathcal{O}(w * 2)$.

The MPI framework allows the registration of custom functions for collective calls, such as `MPI_Scan`. These must be associative and optionally commutative and the above operation satisfies both criteria. However, the reduction operation must also be statically linked, meaning a singleton, and does not allow to pass any context or state, say an object pointer or the aforementioned map. For this reason, a straight forward realization is not possible. One way of working around this limitation is to simply also have only a singular static remapping data-structure in the scope, say a global variable or static class member.

Algorithm 7 Pseudocode of a thread-safe, stateful MPI reduction operation and subsequent usage by a prefix-scan.

```
1: mutex ← CREATE_MUTEX();
2: rules ← CREATE_MAP();
3:
4: procedure REDUCTIONOPERATION(in, out)
5:     LOCK(mutex);
6:     local_rules ←FIND(rules, thread_id);
7:     UNLOCK(mutex);
8:     MERGE(in, out, local_rules);            ▷ actual work
9:
10: procedure CAPTURESTATE(local_rules)
11:     LOCK(mutex);
12:     PUT(rules, local_rules);
13:     UNLOCK(mutex);
14:
15: local_rules ←CREATE_RULES();
16: op ←MPI_OP_CREATE(REDUCTIONOPERATION);
17: CAPTURESTATE(local_rules);
18: MPI_SCAN(..., op);
```

In this case, though, the whole distributed max-tree implementation effectively also becomes a singleton and may not be used in multi-threaded environments, which is sub-optimal for a number of analysis uses cases. Therefore, it has been chosen an approach as sketched in Algorithm 7. A set of potential remapping data-structures from different threads is stored in a global associative container, here $rules$, guarded by a $mutex$. Before calling an `MPI_Scan` the remapping data-structure must be stored and during the custom reduction operation retrieved. In order to be able to correctly retrieve the remapping data-structure a unique, shared key must be chosen, for example the current thread identifier.

One enhancement for future MPI standard versions could be, to directly allow the possibility of passing context pointers to every MPI API call, which utilizes reduction operations. This pointer is simply forwarded to the custom reduction operation on each invocation and can simply be set to undefined, respectively null, if not needed. As a result, this would remove the locking overhead and the code becomes cleaner and more understandable.

6 Experimental Evaluation

6.1 Environment

The experiments have been performed on the JURECA system [49] at the Juelich Supercomputing Centre. The system consists of 1884 compute nodes with each having two *Intel® Xeon® E5-2680 v3 Haswell* CPUs with 12 cores at 2,5 GHz and Hyperthreading. 1604 compute nodes have 128 GiB, 128 nodes 256 GiB, 74 node 512 GiB and two nodes 1024 GiB DDR5 RAM. For our experimental evaluation the following software libraries have been used—`HDF5 1.8.18 parallel` and `ParaStation MPI 5.1.9`. Source code has been compiled with `g++ 5.4.0` optimization level `O3`. The available benchmark for the experiments relies on a maximum of 32 nodes and 24 threads.

6.2 Datasets

As for the used data, the tests have been performed on two real-world images depicted in Figure 4. The first dataset is a Pléiades Ortho Product [1] and it includes four Pan-sharpened images. The spatial and radiometric resolution is 0.5m and 8 bpp, respectively. The data was acquired over the Naples metropolitan area (Italy) in 2013. The second dataset is an image that was taken at the ESO Paranal Observatory in Chile by the VISTA infra-red wide-field survey telescope. It portraits more than 84 million stars in the central regions of the Milky Way [50]. The Figure 4a and Figure 4b show the true-color image of both datasets. For the Naples dataset, experiments are performed only using the first channel. For the ESO, the RGB image is simplified to a singular luminance channel, similarly how it was done by Moschini *et al.* [36], through weighing and summing the channels, so that $L = 0.2126R + 0.7152G + 0.0722B$. However, in order to show that the algorithm scales regardless of the domain size of the gray-levels, three different quantization levels are derived from the luminance channel: 8-uint bpp, 16-uint bpp and 32-float bpp. Contrary to [36], the original size of the image is preserved (≈9Gpx), since the JURECA system provides node with large memory.

6.3 Experimental Setup

As discussed in Section 3 there are a number of other serial and parallel versions of the algorithm. Most of them report different value permutations for the computation time, memory consumption, speed-up and scalability of their implementations. Carlinet *et al.* [38] provide their used benchmarks, datasets and the source codes in C++ for many different serial and parallel algorithms[2]. In order to compare results achievable by using serial and parallel computing, the Berger *et al.* [16] algorithm has been selected. Moschini *et al.* [36] proved that Berger is the fastest sequential algorithm for

1. http://www.intelligence-airbusds.com/en/23-sample-imagery
2. https://www.lrde.epita.fr/wiki/Publications/carlinet.14.itip

images with high quantization values and even floating. However the algorithms depend on the MILENA image processing library [51] (i.e., provide fundamental image types and I/O functionality) which was not designed to handle very large images and floating values. For these reasons it was necessary to re-write a new C++ implementation of the algorithm which is library independent. For the parallel processing case, the hybrid shared-memory parallel max-tree algorithm developed by Moschini *et al.* [36] was considered. The algorithm has been implemented in C using POSIX threads and the source code is available publicly[3]. Contrary to the MILENA library, this algorithm has been proposed with the purpose of dealing with large-scale and high-dynamic range images, and was therefore ready to be used out-of-the-box. It may be argued that the comparison is not entirely fair due to the different nature of the algorithms—i.e., shared-memory and distributed-memory—but can very well be investigated for the same number of utilized cores. The expectation naturally is that distributed memory implementation, as the one proposed here, are naturally going to have more overhead compared to shared-memory versions. To the best of our knowledge the only distributed max-tree algorithm has been proposed recently by Kazemier *et al.* [37], but the source code was not obtainable at the time of writing as it is not yet released.

The performance assessment of the algorithms proposed by Berger and Moschini against the algorithm proposed in this paper is conducted with two kinds of benchmarks. The first type is focused on the computation time and speed-up, while the second measures memory consumption. Each benchmark configuration, meaning a particular node and core count, is executed five times and the following statistics are reported: mean μ, standard deviation σ, minimum, maximum, and coefficient of variation (CV), defined as $\nu = \frac{\sigma}{\mu}$ [52]. The use of the multithreading/MPI hybrid features of the algorithm allows to span the MPI process on each node available and to parallelize it locally using multithreading. For this reason, both types of benchmarks are performed on each number of cores, as shown in Table 2. The strategy is to evaluate first the performance on one core of one node. Afterwards the number of threads are doubled alternating with doubling the number of nodes, until the maximum of 256 cores across nodes and threads is reached.

TABLE 2: Hybrid: multithreading+MPI.

Nodes	1	1	2	2	4	4	8	8	16
Threads	1	2	2	4	4	8	8	16	16
Cores	1	2	4	8	16	32	64	128	256

6.4 Speed-up and Memory Consumption

The Figure 5 depicts the experimental results related to the processing time. For each dataset, the plot of the mean execution time and the plot of the speed-up for increasing number of cores is reported. In order to make a fair comparison with the state-of-the-art results (i.e., the shared-memory algorithm [36]), the proposed algorithm is first run on a single node. The algorithm's execution time measures the beginning and end of the `main()` function of the process with the MPI rank 0 and the thread number 0. The speed-up coefficient is computed as $t_p = t_1/t_c$. the fraction of the execution time with a single core and the execution time with multiple processing cores. Generally it can be said that the proposed algorithm is able to gain a substantial speed-up for both data sets and the different gray-levels quantizations.

For the 8bpp case, the algorithm shows a constant, near linear speed-up curve. In both datasets, the speed-up shows an increasing behavior for up to 256 cores with no reason to doubt its consistency for a higher number of cores, with an execution time of 9.38 and 62.35 seconds for Naples and ESO, respectively. However, for ESO 16bpp and 32bpp the speed-up flattens sooner, stabilizing at 64 and 16 cores, respectively. With a high gray-level depth, the number of tuples is increasing sharply, resulting in larger merge time. The effect observable here is the Amdahl speed-up boundary for a constant workload.

When these results are compared with the Moschini algorithm, the proposed algorithm always provides faster execution times. Unfortunately, for the dataset ESO 32bpp it was not possible to derive any conclusions since the Moschini algorithm did not terminate. A more detailed analysis of the execution time for the different phases of the algorithm (see the Algorihtm 2) is depicted in Figure 6. The results are related only to a single dataset case (ESO 16bpp) because of space considerations. Each set of rows depicts a a specific number of nodes (i.e., 1 node, 2 nodes, 4 nodes and 8 nodes). The time distribution for increasing number of threads is shown in each row. For the single node case, the computation of the local max tree is the most time-consuming phase. This is a shared-memory scenario where the three phases concerning the management of the tuples do not take place. When the number of threads increases, the second most costly phase is the local apply. The local merge needs to be considered only beyond four threads. The same conclusions can be derived for the remaining nodes configuration. However, since it is a distributed memory environment, the phases connected with tuple handling are also present.

In the two-node case the tuple generation phase and the global apply are mostly present, the weight of the tuple resolution becomes more pronounced with a higher number of nodes. This behavior can best be explained by the algorithm complexity, explained in Table 1, showing a logarithmic scaling with the number of processing cores. As has been shown here, the parallel implementation allows to achieve a significant processing time gain when compared with serial processing. In Table 3 the processing times for the different datasets are presented. When considering ESO 32bpp, which is the more challenging dataset used in this work, the proposed algorithm computes the max-tree in ≈27 minutes (with 24 cores) while Berger converges only after ≈3 hours.

Last but not least, comments should be made regarding the memory consumption of the proposed algorithm. Considering the usual trade-off between memory consumption and computational time, the experiments show the proposed algorithm is more memory efficient and takes shorter computational time than Berger and Moschini. Table 4 and Table 5 scrutinize the memory consumption (in GB) for the different

3. http://www.cs.rug.nl/ michael/ParMaxTree/

algorithms with Naples and ESO 16bpp images, respectively, when computing on a single node. For each given number of threads, it can be noticed that the average and the maximum memory usage of all the tasks in the job are always lower for the proposed algorithm. This means that the algorithm is able to scale in terms of memory consumption and communication cost with respect to large datasets and the number of parallel cores. This is an important factor, considering most of the time the main constraint lies in the memory size.

TABLE 3: Processing times (mean values in minutes and statistics) of the sequential Berger algorithm.

Images	Mean μ	StDev σ	CV	Min	Max
Naples	13.54	0.484	0.001	13.53	13.55
ESO 8bpp	113.76	6.011	0.001	113.62	113.91
ESO 16bpp	184.56	9.877	0.001	184.35	184.73
ESO 32bpp	185.67	19.457	0.002	185.34	186.21

TABLE 4: Memory consumption (mean values in Gb and statistics) for the different algorithms with Naples image when using a single node. For each threads setup, the average and the maximum resident set size of all the tasks in the job are reported, respectively.

Algorithm	Threads	Mean μ	StDev σ	CV	Min	Max	
Berger *et al.* [16]	1	59.65	0.327	0.005	59.35	60.19	Average
		66.89	6.429	0.096	63.29	78.13	Maximum
Moschini *et al.* [36]	1	91.99	1.955	0.021	89.64	93.52	Average
		116.83	0.000	0.000	116.83	116.83	Maximum
	2	89.03	0.590	0.007	88.10	89.64	Average
		116.83	0.000	0.000	116.83	116.83	Maximum
	4	82.82	1.988	0.024	81.19	86.09	Average
		116.83	0.000	0.000	116.83	116.83	Maximum
	8	79.33	3.161	0.040	73.99	81.80	Average
		116.84	0.000	0.000	116.84	116.84	Maximum
	16	74.23	5.571	0.075	64.96	79.14	Average
		116.84	0.000	0.000	116.84	116.84	Maximum
	24	62.45	4.593	0.074	58.83	70.03	Average
		116.84	0.000	0.000	116.84	116.84	Maximum
Proposed	1	18.93	0.340	0.018	18.43	19.32	Average
		22.36	0.106	0.005	22.22	22.45	Maximum
	2	18.40	0.657	0.036	17.81	19.12	Average
		23.13	0.106	0.005	22.94	23.18	Maximum
	4	18.18	0.107	0.006	18.06	18.33	Average
		22.92	0.178	0.008	22.68	23.17	Maximum
	8	16.23	1.159	0.071	14.79	17.19	Average
		23.07	0.544	0.024	22.44	23.64	Maximum
	16	14.28	2.082	0.146	10.66	15.63	Average
		24.24	0.686	0.028	23.24	25.15	Maximum
	24	23.11	1.872	0.081	20.42	25.00	Average
		22.35	0.099	0.004	22.18	22.41	Maximum

7 Conclusion

In this work a new parallel and distributed algorithm for the computation of the max-tree of an image has been presented. The parallelization strategy consists of splitting the entire problem, i.e., the image, into equal-sized sub images, for which the partial max-trees are computed that are subsequently merged at the split boundaries. Using this algorithm, substantial speed-ups and scalability could be achieved in computing the max-tree on large real-world images, outperforming the state-of-the-art shared memory implementation. In particular, faster execution time and significantly less memory consumption can be achieved. The proposed algorithm allows to process gray-scale image of arbitrary gray-level depth including floating point values. This makes it suitable for the usage in large-scale image classification task, such as land cover type prediction, which is one of the major practical application domains.

TABLE 5: Memory consumption (mean values in Gb and statistic) for the different algorithms with ESO 16bpp image when using a single node. For each threads setup, the average and the maximum resident set size of all the tasks in the job are reported.

Algorithm	Threads	Mean μ	StDev σ	CV	Min	Max	
Berger *et al.* [16]	1	292.30	0.217	0.001	291.93	292.48	Average
		296.70	0.001	0.000	296.70	296.70	Maximum
Moschini *et al.* [36]	1	457.71	0.777	0.002	456.98	458.61	Average
		525.21	0.001	0.000	525.21	525.21	Maximum
	2	472.53	0.898	0.002	471.54	473.76	Average
		525.21	0.001	0.000	525.21	525.21	Maximum
	4	436.67	1.761	0.004	434.06	437.89	Average
		525.21	0.001	0.000	525.21	525.22	Maximum
	8	409.87	3.323	0.008	406.41	414.84	Average
		525.21	0.001	0.000	525.21	525.22	Maximum
	16	383.99	2.466	0.006	381.68	387.10	Average
		525.22	0.001	0.000	525.22	525.22	Maximum
	24	380.92	5.482	0.014	376.22	388.25	Average
		525.22	0.001	0.000	525.22	525.22	Maximum
Proposed	1	102.39	0.091	0.001	102.32	102.54	Average
		114.13	0.052	0.000	114.08	114.20	Maximum
	2	103.04	0.307	0.003	102.61	103.43	Average
		113.84	0.028	0.000	113.81	113.88	Maximum
	4	102.34	0.528	0.005	101.65	103.05	Average
		117.99	0.036	0.000	117.95	118.03	Maximum
	8	100.80	0.458	0.005	100.25	101.46	Average
		121.55	0.068	0.001	121.48	121.62	Maximum
	16	94.86	1.264	0.013	93.58	96.30	Average
		124.43	0.185	0.001	124.27	124.72	Maximum
	24	94.27	1.714	0.018	92.95	96.19	Average
		124.82	0.241	0.002	124.46	125.05	Maximum

In future work, the equivalent min-tree algorithm including distributed attribute filter are going to be implemented. This will set a solid foundation for the next research goal, the massive parallelization of the tree of shapes [53]—a contrast independent component tree representation of images.

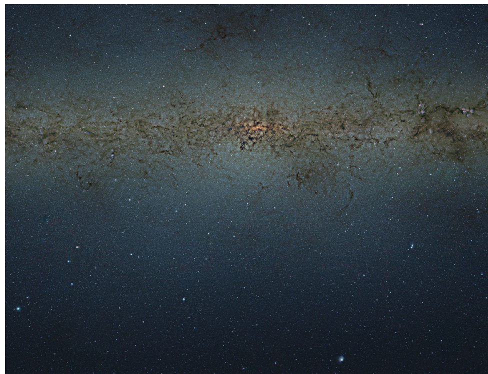
Acknowledgments

The authors would like to thank Igancio Toledo and Martin Kornmesser for making the ESO/VVV Survey/D. Minniti image with the id *eso1242a* publicly available.

References

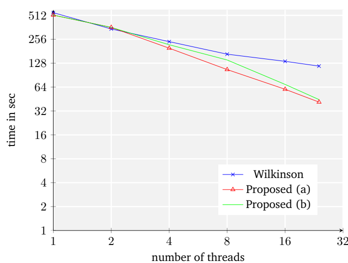
[1] G. Matheron, *Random Sets and Integral Geometry*. New York: John Wiley & Sons, 1975.
[2] J. Serra, *Image Analysis and Mathematical Morphology*. London: Academic Press, 1982.
[3] E. J. Breen and R. Jones, "Attribute Openings, Thinnings, and Granulometries," *Computer Vision and Image Understanding*, vol. 64, no. 3, pp. 377–389, 1996.
[4] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive Connected Operators for Image and Sequence Processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, 1998.
[5] L. Najman and J. Cousty, "A Graph-based Mathematical Morphology Reader," *Pattern Recognition Letters*, vol. 47, pp. 3–17, 2014.
[6] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, vol. 7, pp. 48–50, 1956.
[7] G. K. Ouzounis and P. Soille, "The Alpha-Tree Algorithm," *Publications Office of the European Union*, 2012.
[8] P. Salembier and L. Garrido, "Binary Partition Tree as an Efficient Representation for Image Processing, Segmentation, and Information Retrieval," *IEEE Transactions on Image Processing*, vol. 9, no. 4, pp. 561–576, 2000.
[9] R. Jones, "Component Trees for Image Filtering and Segmentation," in *Procedings of the IEEE Workshop on Nonlinear Signal and Image Processing (NISP)*, E. Coyle, Ed., 1997.
[10] V. Caselles, B. Coll, and J. M. Morel, "Topographic Maps and Local Contrast Changes in Natural Images," *International Journal of Computer Vision*, vol. 33, no. 1, pp. 5–27, 1999.

(a) Pan-sharpened, true-color image of Naples, Italy.

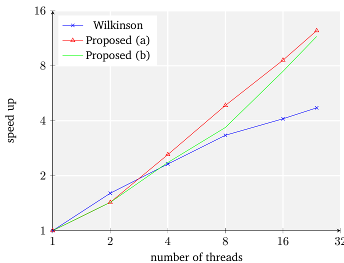


(b) Center region of the Milky Way, ESO, Chile.

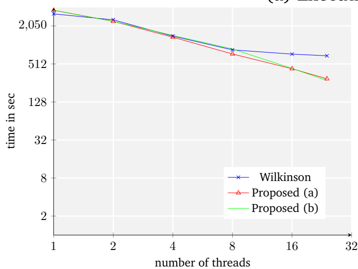
Fig. 4: Benchmark images used in the experimental evaluation of the algorithm.



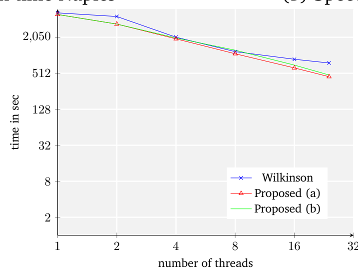
(a) Execution time Naples



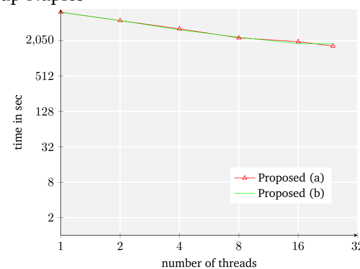
(b) Speed-up Naples



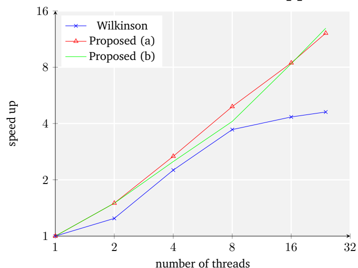
(c) Execution time ESO 8bpp



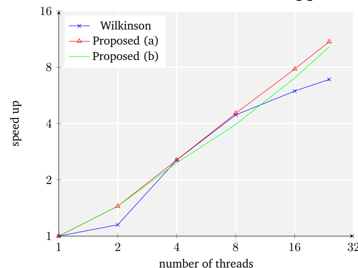
(d) Execution time ESO 16bpp



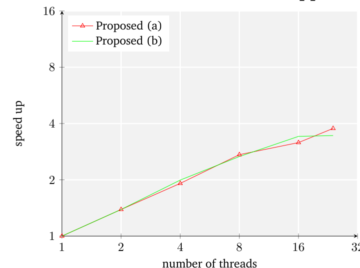
(e) Execution time ESO 32bpp



(f) Speed-up ESO 8bpp



(g) Speed-up ESO 16bpp



(h) Speed-up ESO 32bpp

Fig. 5: Execution time and speed-up curves of the Moschini and the proposed algorithm for increasing number of cores. The Moschini and the Proposed (Multithreading) algorithm are run on a single node with increasing number of threads. The proposed (Hybrid) is run with a serie of increasing number of nodes and threads as shown in Table 2.

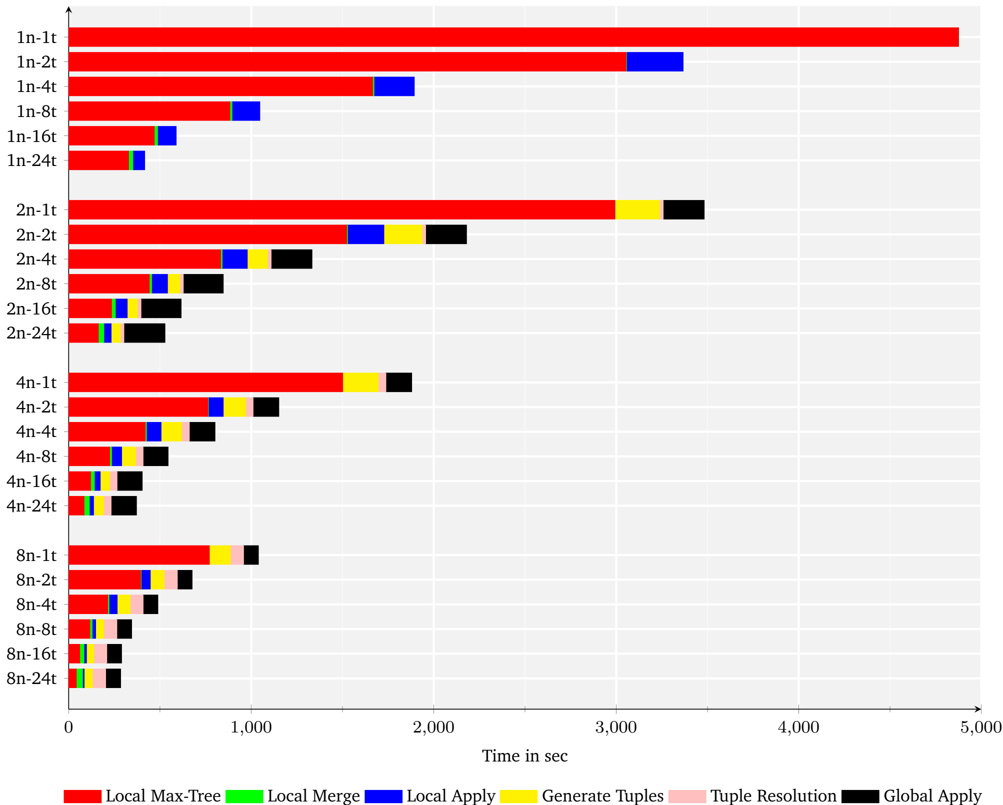


Fig. 6: Execution time distribution of the proposed algorithm for the ESO 16bpp dataset. The notation: n is number of nodes and t is number of threads.

[11] E. Carlinet and T. Géraud, "MToS: A Tree of Shapes for Multivariate Images," *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5330–5342, 2015.

[12] P. Salembier and J. Serra, "Flat Zones Filtering, Connected Operators, and Filters by Reconstruction," *IEEE Transactions on Image Processing*, vol. 4, no. 8, pp. 1153–1160, 1995.

[13] G. K. Ouzounis and M. H. F. Wilkinson, "Mask-Based Second-Generation Connectivity and Attribute Filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 990–1004, 2007.

[14] M. Dalla Mura, J. A. Benediktsson, B. Waske, and L. Bruzzone, "Morphological Attribute Filters for the Analysis of Very High Resolution Remote Sensing Images," in *IEEE International Geoscience and Remote Sensing Symposium (IGARSS '09)*, vol. 3, 2009, pp. 2–3.

[15] J. A. Benediktsson, L. Bruzzone, J. Chanussot, M. Dalla Mura, P. Salembier, and S. Valero, "Hierarchical Analysis of Remote Sensing Data: Morphological Attribute Profiles and Binary Partition Trees," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6671 LNCS, 2011, pp. 306–319.

[16] C. Berger, T. Geraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin, "Effective Component Tree Computation with Application to Pattern Recognition in Astronomical Imaging," in *IEEE International Conference on Image Processing*, 2007, pp. IV – 41–IV – 44.

[17] P. Teeninga, U. Moschini, S. C. Trager, and M. H. F. Wilkinson, "Improved Detection of Faint Extended Astronomical Objects Through Statistical Attribute Filtering," in *Mathematical Morphology and Its Applications to Signal and Image Processing (ISMM): 12th International Symposium*, J. A. Benediktsson, J. Chanussot, L. Najman, and H. Talbot, Eds. Springer International Publishing, 2015, pp. 157–168.

[18] I. K. E. Purnama, K. Y. E. Aryanto, and M. H. F. Wilkinson, "Non-Compactness Attribute Filtering to Extract Retinal Blood Vessels in Fundus Images," *International Journal of E-Health and Medical Communications (IJEHMC)*, vol. 1, no. 3, pp. 16–27, 2010.

[19] F. N. Kiwanuka and M. H. F. Wilkinson, "Automatic Attribute Threshold Selection for Morphological Connected Attribute Filters," *Pattern Recognition*, vol. 53, no. C, pp. 59–72, 2016.

[20] G. Cavallaro, N. Falco, M. D. Mura, and J. A. Benediktsson, "Automatic Attribute Profiles," *IEEE Transactions on Image Processing*, vol. 26, no. 4, pp. 1859–1872, 2017.

[21] B. Song, M. Dalla Mura, P. Li, A. Plaza, J. M. Bioucas-Dias, J. A. Benediktsson, and J. Chanussot, "Remotely Sensed Image Classification Using Sparse Representations of Morphological Attribute Profiles," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 52, no. 8, pp. 5122–5136, 2014.

[22] N. Falco, J. A. Benediktsson, and L. Bruzzone, "Spectral and Spatial Classification of Hyperspectral Images Based on ICA and Reduced Morphological Attribute Profiles," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 53, no. 11, pp. 6223–6240, 2015.

[23] M. Pedergnana, P. R. Marpu, M. Dalla Mura, J. A. Benediktsson, and L. Bruzzone, "Classification of Remote Sensing Optical and LiDAR Data Using Extended Attribute Profiles," *IEEE Journal of Selected Topics in Signal Processing*, vol. 6, no. 7, pp. 856–865, 2012.

[24] N. Falco, M. Dalla Mura, F. Bovolo, J. A. Benediktsson, and L. Bruzzone, "Change Detection in VHR Images Based on Morphological Attribute Profiles," *IEEE Geoscience and Remote Sensing Letters*, vol. 10, no. 3, pp. 636–640, 2013.

[25] G. K. Ouzounis, M. Pesaresi, and P. Soille, "Differential Area Pro-

files: Decomposition Properties and Efficient Computation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 8, pp. 1533–1548, 2012.

[26] G. Ouzounis and M. Wilkinson, "A parallel dual-input max-tree algorithm for shared memory machines," in *Mathematical Morphology and Its Applications to Signal and Image Processing (ISMM): 8th International Symposium*, 2007, pp. 449–460.

[27] P. Flick, C. Jain, T. Pan, and S. Aluru, "A Parallel Connectivity Algorithm for De Bruijn Graphs in Metagenomic Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York: ACM, 2015, pp. 1–11.

[28] R. Jones, "Connected Filtering and Segmentation Using Component Trees," *Computer Vision and Image Understanding*, vol. 75, no. 3, pp. 215–228, 1999.

[29] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *ACM Journal*, vol. 22, no. 2, pp. 215–225, 1975.

[30] L. Najman and M. Couprie, "Building the component tree in quasi-linear time," *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3531–3539, 2006.

[31] W. H. Hesselink, "Salembier's min-tree algorithm turned into breadth first search," *Information Processing Letters*, vol. 88, no. 5, pp. 225 – 229, 2003.

[32] D. Nistér and H. Stewénius, "Linear time maximally stable extremal regions," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5303 LNCS, no. PART 2, 2008, pp. 183–196.

[33] M. H. F. Wilkinson, "A fast component-tree algorithm for high dynamic-range images and second generation connectivity," in *Proceedings - International Conference on Image Processing, ICIP*, 2011, pp. 1021–1024.

[34] M. H. F. Wilkinson, H. Gao, W. H. Hesselink, J. E. Jonker, and A. Meijster, "Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 10, pp. 1800–1813, 2008.

[35] G. K. Ouzounis and L. Gueguen, "Interactive collection of training samples from the max-tree structure," in *18th IEEE International Conference on Image Processing*, 2011, pp. 1449–1452.

[36] U. Moschini, A. Meijster, and M. Wilkinson, "A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PP, no. 99, pp. 1–1, 2017.

[37] J. J. Kazemier, G. K. Ouzounis, and M. H. F. Wilkinson, *Connected Morphological Attribute Filters on Distributed Memory Parallel Machines*. Springer International Publishing, 2017, pp. 357–368.

[38] E. Carlinet and T. Géraud, "A Comparative Review of Component Tree Computation Algorithms," *IEEE Transactions on Image Processing*, vol. 23, no. 9, pp. 3885–3895, 2014.

[39] P. Matas, E. Dokládalová, M. Akil, T. Grandpierre, L. Najman, M. Poupa, and V. Georgiev, "Parallel algorithm for concurrent computation of connected component Tree," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5259 LNCS, 2008, pp. 230–241.

[40] H. Samet and M. Tamminen, "Efficient component labeling of images of arbitrary dimension represented by linear bintrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 4, pp. 579–586, 1988.

[41] J. Iverson, C. Kamath, and G. Karypis, "Evaluation of connected-component labeling algorithms for distributed-memory systems," *Parallel Computing*, vol. 44, pp. 53–68, 2015.

[42] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.

[43] N. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Akademie Van Wetenschappen*, vol. 49, no. 6, pp. 758–764, 1946.

[44] P. Soille, *Morphological Image Analysis: Principles and Applications*, 2nd ed. Springer-Verlag Berlin Heidelberg New York, 2004.

[45] M. Goetz, G. Cavallaro, T. Geraud, M. Book, and M. Riedel, "Distributed Max-Tree," https://www.codeocean.com/, September 2017. [Online]. Available: https://codeocean.com/algorithm/8ceb5734-5ca5-4665-a415-a50a5d9b8bec/code

[46] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*, 2000, vol. 40, no. 2-3.

[47] HDF Group, "Hierachical Data Format 5." [Online]. Available: http://www.hdfgroup.org/HDF5

[48] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[49] Jülich Supercomputing Centre, "JURECA: General-purpose supercomputer at Jülich Supercomputing Centre," *Journal of large-scale research facilities*, vol. 2, no. A62, 2016.

[50] R. K. Saito, D. Minniti, B. Dias, M. Hempel, M. Rejkuba, J. Alonso-García, B. Barbuy, M. Catelan, J. P. Emerson, O. A. Gonzalez, P. W. Lucas, and M. Zoccali, "Milky Way demographics with the VVV survey," *Astronomy & Astrophysics*, vol. 544, 2012.

[51] R. Levillain, T. Géraud, and L. Najman, "Why and how to design a generic and efficient image processing framework: The case of the Milena library," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, Hong Kong, 2010, pp. 1941–1944.

[52] H. Abdi, "Coefficient of variation," *Encyclopedia of research design*, pp. 169–171, 2010.

[53] S. Crozet and T. Géraud, "A first parallel algorithm to compute the morphological tree of shapes of nD images," in *Proceedings of the 21st IEEE International Conference on Image Processing (ICIP)*, Paris, France, 2014, pp. 2933–2937.

Markus Götz received his Bachelor of Science degree in Software Engineering from Hasso-Plattner-Institute, University of Potsdam, Potsdam, Germany in 2010. In 2014 he has been awarded with a Master of Science degree in Software Engineering from the same institution. During this time has gathered experience in data analysis, image processing and data mining during his stays at Blekinge Tekniska Högskola, Sweden, the European Organization for Nuclear Research (CERN), Switzerland and mental images GmbH, Germany. Currently, he is with the Juelich Supercomputing Center, Germany and the University of Iceland in line with his Ph.D. studies. His research interests include high-performance computing, parallel algorithms, machine learning as well as time series and data analysis.

Gabriele Cavallaro received the B.S. and M.S. degrees in telecommunications engineering from the University of Trento, Italy, in 2011 and 2013, respectively. He holds a Ph.D. degree in Electrical and Computer Engineering from the University of Iceland, obtained in 2016. At present he is a postdoctoral research assistant at the Juelich Supercomputing Centre, Juelich, Germany. At this institute, he is part of a scientific research group focused on high productivity data processing within the Federated Systems and Data Division. His research interests include remote sensing and analysis of very high geometrical and spectral resolution optical data with the current focus on mathematical morphology and high performance computing. He was the recipient of the IEEE GRSS Third Prize in the Student Paper Competition of the 2015 IEEE International Geoscience and Remote Sensing Symposium 2015 (Milan, Italy, July 2015). He serves as a reviewer for IEEE Geoscience and Remote Sensing Letters and IEEE Journal of Selected Topics in Earth Observations and Remote Sensing.

Thierry Géraud received a Ph.D. degree in signal and image processing from Télécom ParisTech in 1997, and the Habilitation à Diriger les Recherches from Université Paris-Est in 2012. He is one of the main authors of the Olena platform, dedicated to image processing and available as free software under the GPL licence. His research interests include image processing, pattern recognition, software engineering, and object-oriented scientific computing. He is currently working at EPITA Research and Development Laboratory (LRDE), Paris, France.

Matthias Book Matthias Book is professor for software engineering at the University of Iceland. After receiving his doctoral degree from the University of Leipzig, he worked as Research Manager for the German software company adesso AG, led the Mobile Interaction group at the University of Duisburg-Essen's Ruhr Institute for Software Technology (paluno), and served as acting head of the Software Engineering and Information Systems Chair at Chemnitz University of Technology. His research focus is on facilitating collaboration between domain experts and technology experts in complex software projects.

Morris Riedel received his Ph.D. degree from Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, and started working in parallel and distributed systems in the field of scientific visualization and computational steering of e-science applications on large-scale HPC resources. He is an Adjunct Associate Professor with the School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland. He previously held various positions at the Juelich Supercomputing Centre, Juelich, Germany. At this institute, he is also the head of a scientific research group focused on high productivity data processing as a part of the Federated Systems and Data Division. The given lectures in universities such as the University of Iceland, University of Applied Sciences of Cologne, Cologne, Germany, and the University of Technology Aachen (RWTH Aachen), Aachen, Germany include High Performance Computing and Big Data, Statistical Data Mining, and Handling of large Datasets and Scientific and Grid Computing. His research interests include high productivity processing of big data in the context of scientific computing applications.

Appendix F

Paper VI

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SUPERCOMPUTING'17, November 2017, Denver, Colorado, USA
© 2017 Association for Computing Machinery.
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. 2017. 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. [4] that is additionally supported by research of Segal and Morris [32] and Schmidberger and Brügge [31].

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 [32]. 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 [31].

Finally, the ranking of engineering goals in HPC differs from the engineering of information systems. Carver et al. [7] point out that first and foremost the correctness of the code matters, followed by performance/scalability, portability and maintainability, in that order. The correctness of code in HPC has traditionally been ensured through formal correctness proofs, like partial correction assertions. Given the increasing complexity of applications, this has become generally infeasible. 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 experimentation, such as the Large Hadron Collider [19] or the planned Square Kilometer Array [12] have recently led to a steep increase in interest and requirements. Typical analysis goals are the 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.

Since the 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 such as being the benchmark suite for the DEEP-EST experimental computing system, but also for land cover classification, which will be 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) [26], field-programmable arrays (FPGAs) [28] or Many Integrated Cores (MICs) [17] 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 [36], 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) [15], 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 [20], 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 [24], the usage of agile development processes [34] or studies of performance in comparison to maintainability and scalability [29]. 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 [2]. The awareness of the need 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. When boiling them down based on their capabilities and ability to scale on HPC systems, two frameworks remain—MLPack [11] and Intel DAAL [17]. Both of them follow good software engineering practices in their own source code, which include among others the usage of open source code repositories, an extensive documentation with examples, and the modularization into conceptual components. For an application developer, the object-oriented design assists the quick development of analysis tools that exploit parallelism in C++. Both MLPack and DAAL have their own specific strong suits, which 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. This is due to the fact that MLPack does not feature any data distribution as part of the framework and but leaves this to the developer to implement. Instead, only CPU multi-core parallelism is included and implicitly tested. 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 is the most similar to JuML in terms of goals, features and design. Both have been in development concurrently and Intel's product has been released slightly prior to JuML. DAAL allows the development of data analysis applications in C++, as well as 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 can generally not be used on the same cluster system. The data distribution code with MPI as communication framework needs to be provided by the application user [1], which is often the most error-prone and time-consuming part of HPC application development. Moreover, DAAL only supports Intel's own MICs and does not allow the computation offloading to other coprocessors. Therefore, it is only of limited use in HPC data analysis application development.

Independent of data analysis frameworks, there is a highly productive HPC software library called *ArrayFire* [21]. It offers a high-level vector, matrix and tensor abstraction that allows the execution of computation kernels on different so-called backends. This means processing can be executed on CPUs, a GPU using the CUDA interface or via the OpenCL interface on any other supported coprocessor (though optimized for GPU vector processing). While it is not able to perform distributed computations, it simplifies the development of single-node applications. ArrayFire includes a comprehensive list of highly optimized standard computation routines that further shortens code and development time. It is a good base for application development in heterogeneous cluster systems.

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 [3]. 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) [16] 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

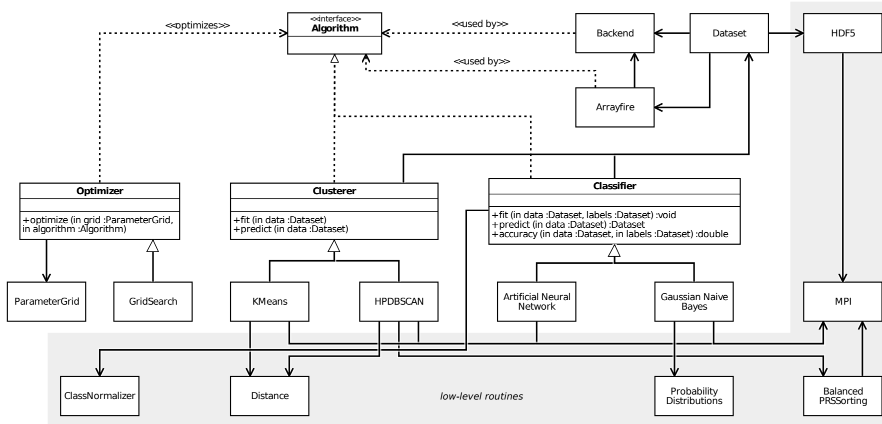


Figure 1: UML diagram of JuML's architecture.

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 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 [23] 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 [13] 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 [27] or SHOGUN [35]. 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.

```
1  #include <juml.h>
2  #include <mpi.h>
3
4  int main(void) {
5      juml::GaussianNaiveBayes gnb(
6          juml::Backend::GPU, // local gpu backend
7          MPI_COMM_WORLD // select global node allocation
8      );
9      return 0;
10 }
```

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 [5] 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.

```
1  import juml
2  from mpi4py import MPI
3
4  gnb = juml.GaussianNaiveBayes(
5      juml.Backend.CPU, # local cpu backend
6      MPI.COMM_SELF # global node allocation
7  )
```

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 [6] or the parallel HPDBSCAN [14] 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* [33]. 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 [31] and in details through the application of effective testing methods [24], mocking approaches [9] or concrete case studies [25]. 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 [37] 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 [38].

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.

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.

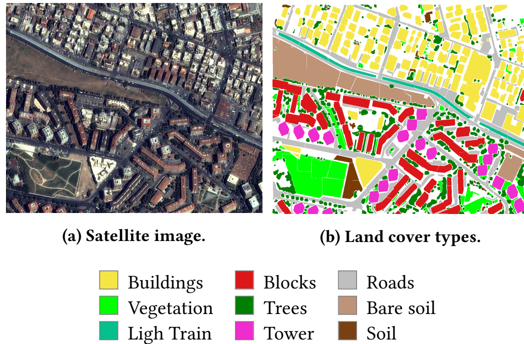
Furthermore, JuML provides an extension to the test runner CTest [22] that is uses to execute the tests distributed on the cluster system. The added ADD_MPI_TEST function again registers an individual test for each of the used node counts, which enables better error tracing. In this function, common 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.

```
1   // original fixture class of Google Test
2   class FIXTURE_TEST {
3       FIXTURE_TEST() {
4           // setup code here
5       }
6   }
7
8   // forward definition inheriting from the fixture
9   #define INTECEPTOR_FORWARD_DEFINITION(FIXTURE) \
10  class FIXTURE##_Interceptor : public FIXTURE { \
11      protected: \
12      void test(); \
13  };
14
15  // per-backend test generator
16  #define TEST_BACKEND_F(FIXTURE, BACKEND) \
17  TEST_F(FIXTURE##_Interceptor) { \
18      // set backend
19      juml::setBackend(BACKEND); \
20      // call the test case
21      this->test(); \
22  }
23
24  // definition of the main macro for all backends
25  #define TEST_ALL_F(FIXTURE) \
26  TEST_INTERCEPTOR_FORWARD_DEFINITION(FIXTURE) \
27  TEST_BACKEND_F(FIXTURE, juml::Backend::CPU) \
28  #ifdef OTHER_BACKEND \
29  TEST_BACKEND_F(FIXTUREm OTHER_BACKEND) \
30  #endif \
31  void FIXTURE##_Interceptor::test
32
33
34  TEST_ALL_F(FIXTURE_TEST, FEATURE) {
35      // test code here
36  }
```

Listing 4: TEST_ALL_F macro implementation sketch.

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 [30]. 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. [8]. The experimentation was performed on the JURECA supercomputer [18] using multiple GPGPU compute nodes, each having two CUDA-aware [26] NVIDIA K80s. Figure 3 shows the obtained speed-ups during the training phase with a batch size of 100.

Another example for JuML's application is the benchmarking of the data analysis hardware module of the experimental DEEP-EST supercomputing system [10]. 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 [14], to be ported to the new platform. Second, and even more important, it also significantly

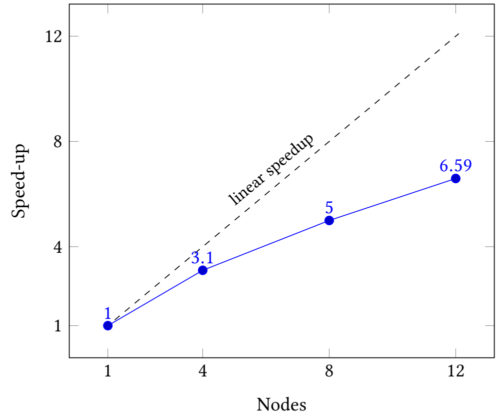
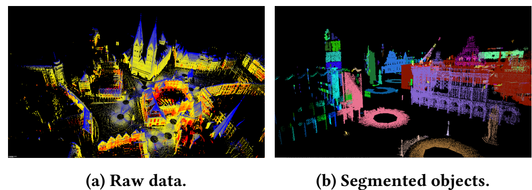


Figure 3: ANN Speedup with 1000 hidden neurons on the Rome land cover type classification problem.

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 parallaleliza-tion 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] 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).

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

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

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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

[19] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. 2014. Trends in big data analytics. *J. Parallel and Distrib. Comput.* 74 (2014), 2561–2573.

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

[21] 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.*

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

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

[31] 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.

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

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

[34] 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.

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

[36] 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.

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

[38] 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.