Carinthia University of Applied Sciences
School of Engineering & IT
Department of Geoinformation and
Environmental Technologies
Spatial Information Management

MASTER THESIS

SOMatic Trainer 2.0: Improved Parallelization Concepts and GeoJSON Integration for a Self-Organizing Map Java Implementation

Submitted in partial fulfillment of the requirements of the academic degree
Master of Science in Engineering

Author: Fabian Georg Kowatsch, BSc.
Registration Nr: 1510362004

Internal Supervisors: FH-Prof. Dr.-Ing. Karl-Heinrich Anders
Carinthia University of Applied Sciences, Villach, Austria

FH-Prof. Mag. Dr. Gernot Paulus, MSc. MAS
Carinthia University of Applied Sciences, Villach, Austria

External Supervisor: Prof. Dr. André Skupin
San Diego State University, San Diego, CA, USA

Villach, September 2017

Abstract

This research is focused on improving a Self-Organizing Map (SOM) Java implementation called SOMatic, precisely the SOMatic trainer (Spöcklberger, 2013). The main parts of this enhancement are focused on decreasing computation time through increasing the parallelization and updating the technologies as well as enlarge the reusability of the output of this Java software. Besides the already implemented local parallelization of the training process, this project explains the development and usage of a parallel search of the best-matching unit (BMU) after the training. This functionality allows the projection of data that was not part of the training onto an existing SOM, which permits a reduction in time as well as ensuring the whole training process does not have to be redone because of the presence of new data that requires visualization. How the SOMatic Trainer 2.0 can be integrated into a distributed system within a network cluster is explained in a conceptual way. This will open the door to train big input data on local network clusters without having the need to seek the help of super computers.

One aspect of updating the status quo from 2013 was to migrate the SOMatic Trainer onto the latest versions of the applied programming language and processing interface, meaning Java 8 and Processing 3. Deploying the code in a Maven project using Java 8 assures that the change onto newer versions of the programming language or required libraries is simplified. To further guarantee proper usage for the future, several comments in the code were added or redefined to better explain the usability of each variable and method. The usage of the SOMatic Trainer is widened via the programmatic integration of the output format "geographic JavaScript object notation" (GeoJSON). This allows one to directly visualize the resulting SOM, as well as the BMUs in a Geographic Information System (GIS). The geometry for each neuron in the resulting GeoJSON file is stored behind six corner points of a hexagon. These points are computed by using the rules of a 30-60-90 triangle. This method was applied to avoid using mathematical functions in the code, which would have a negative impact on the computation time. Two datasets are used to test the SOMatic Trainer 2.0 and show its capabilities: census data from 2001 containing the municipalities of Carinthia, Austria and multispectral data having six spectral bands for three years. The time, as well as the quality measurements on the results of the computations using different threads, show that increasing the parallelization, especially for a bigger dataset, is of a high importance to be able to get the output in a reasonable amount of time without losing too much of the quality.

Keywords: Self-Organizing Map, Parallelization, Java, GeoJSON

Statutory Declaration

I hereby declare that

- this Master Thesis has been written by myself without any external unauthorized help and that it has not been submitted to any institution to achieve an academic grading
- I have not used sources or means without citing them in the text; any thoughts from others or literal quotations are clearly marked
- the enclosed Master Thesis is the same version as the version evaluated by the supervisors
- one copy of the Master Thesis is deposited and made available in the CUAS library (§8 Austrian Copyright Law [UrhG])

I fully understand that I am responsible for the application for a patent, trademark or ornamental design and that I have to prosecute any resultant claims myself.

____

Place and Date Signature

Acknowledgements

I want to thank my family, friends, as well as colleagues and officials from CUAS for their support throughout the five years of my studies in Villach. I got to know several new people from different countries of the world, who also became close friends.

The most important person that helped me to work on and finish this project was Prof. Skupin. Through his talk in Villach, I came in contact with him, the SOM algorithm and its broad usage. Thank you André for your support and for giving me the opportunity to work on a project in your domain at the San Diego State University in California. I also want to thank Tim Schempp, a US student, for his support in our meetings.

Another thank goes to FH-Prof. Karl-Heinrich Anders, who supervised this project during my stay in Villach and gave me critical input to my thesis. Thanks also to FH-Prof. Gernot Paulus for his support, especially at the initial stages of the project.

Without the financial support of the Austrian Marshall Plan Foundation this project would not have been realizable. Thanks to this institution I could afford to conduct my research in the USA.

List of Abbreviations

ANN – Artificial Neural Network

AQE – Average Quantization Error

AWS – Amazon Web Services

BMU – Best-Matching Unit

CPU – Central Processing Unit

CSV – Comma Separated Value

CUAS – Carinthia University of Applied Sciences

GB – GigaByte

GeoJSON – Geographic JavaScript Object Notation

GIS – Geographic Information System

GPU – Graphic Processing Unit

GUI – Graphical User Interface

IDE – Integrated Development Environment

IETF – Internet Engineering Task Force

JDK – Java Development Kit

JPPF – Java Parallel Processing Framework

JRE – Java Runtime Environment

JSON – JavaScript Object Notation

MPI – Message Passing Interface

MR-MPI – Map Reduced Message Passing Interface

OGC – Open Geospatial Consortium

POM – Project Object Model

PVM – Parallel Virtual Machine

QGIS – Quantum GIS

SDSU – San Diego State University

SLA – Service Level Agreement

SMT – Simultaneous MultiThreading

SOM – Self-Organizing Map

SVN – SubVersioN

TST – Training Surveillance Thread

XML – eXtensible Markup Language

Table of Content

Abstract ... i
Statutory Declaration ... ii
Acknowledgements ... iii
List of Abbreviations ... iv
1. Introduction ... 1
1.1 Motivation and Objective ... 1
1.2 Problem Definition ... 3
1.3 Research Questions ... 5
1.4 Methodology ... 6
1.5 Expected Results ... 7
1.6 Thesis Structure ... 8
2. Theoretical Background ... 9
2.1 Self-Organizing Map ... 9
2.2 SOM Algorithm ... 10
2.2.1 Conventional SOM Algorithm ... 10
2.2.2 Batch SOM Algorithm ... 12
2.2.3 Parallelizing the SOM Algorithm ... 14
2.4 Important Languages ... 15
2.4.1 Java ... 15
2.4.2 GeoJSON ... 17
2.5 Distributed Systems ... 18
2.5.1 OpenMPI ... 18
2.5.2 Apache Hadoop ... 18
2.5.3 Java Parallel Processing Framework ... 20
3. Practical Background ... 22
3.1 Parallel SOM Implementations ... 22
3.1.1 Hybrid Parallel SOM Algorithm ... 22
3.1.2 Speculative Parallel SOM Algorithm ... 24
3.1.3 Data Partition Method for Parallel SOMs ... 25
3.2 Network-Distributed SOM Implementations ... 26
3.2.1 Training a SOM distributed on a PVM network ... 26
3.2.2 A GPU accelerated distributed SOM implementation ... 29
3.2.3 Somoclu: A parallel library for SOMs ... 30
3.3 SOMatic ... 32
3.3.1 SOMatic Trainer ... 33
3.3.2 SOMatic Viewer ... 37
4. Method of Solution ... 38
4.1 Procedural Workflow ... 38
4.1.1 Background ... 38

4.1.2 Method of Solution ... 39
4.1.3 Implementation ... 39
4.1.4 Results ... 39
4.2 Software Upgrade ... 39
4.2.1 From Java 6 to Java 8 ... 39
4.2.2 From Processing 2 to Processing 3 ... 40
4.2.3 Code Versioning and Cleanup ... 40
4.3 GeoJSON ... 40
4.4 Parallelization ... 41
4.4.1 Local Parallelization: The BMU Search ... 41
4.4.2 Distributed Parallelization: Cluster Computation in a Network ... 42
4.5 Software, Tools and Libraries ... 44
5. Implementation ... 46
5.1 Working Java Code ... 46
5.2 GeoJSON Output ... 46
5.3 Geometry of the Neurons ... 48
5.4 Integration of a parallelized BMU Search after Training ... 50
5.5 Cluster computing of SOMatic ... 52
5.6 Challenges/Problems ... 52
6. Results ... 55
6.1 SOMatic Trainer 2.0 ... 55
6.2 Applying SOMatic on Test Data ... 57
6.2.1 Carinthian Census Data ... 57
6.2.2 Multispectral Image Data ... 60
6.2.3 Quality and Performance Evaluation ... 61
6.3 Cluster Computation ... 69
7. Discussion and Conclusion ... 70
8. Future Work ... 73
References ... 75
List of Figures ... 78
List of Tables ... 79

1. Introduction

Enhancing a software, in this case, means to enlarge its functionality and productivity by improving the existing components and adding new ones. This project focuses on a SOM Java implementation called SOMatic which has been expanded. The Self-Organizing Map algorithm is a data mining technique to reduce the dimensionality of data and cluster it to get a better understanding of it. It can be applied on normalized data to be able to find relations and differences within the data that one might not have been able to detect beforehand. Through creating visualizations with GI-software, these characteristics of the data can be examined. Java is the programming language in which SOMatic and the cluster computing software are written.

1.1 Motivation and Objective

One important aspect of today's data mining approaches is to handle the increase in data size and its complexity. Especially image data or photos can be considered as Big Data (Tole, 2013) because each pixel holds a certain amount of information and the better the resolution, the higher the number of pixels. The SOM is a useful dimensionality reduction tool when applied to multispectral or hyperspectral image data (Jordan & Angelopoulou, 2013), as it can yield a better understanding of the data. Therefore, it is useful to create and improve software, which uses this technique. The SOM algorithm can be a useful tool to get from high-dimensional data to two-dimensions through applying a reduction of the dimensions of the input data and clustering it. It minimizes the size as well as the complexity to be able to have a clearer view on the relations and differences within the data. SOMatic is a Java implementation of this algorithm that was developed by a joint student research project between the San Diego State University (SDSU) and the Carinthia University of Applied Sciences (CUAS) in 2013. Spöcklberger (2013) created the SOMatic Trainer, an implementation of a SOM tool with parallel training usable as Java library for Processing or as standalone Java program. Rainer (2013) created the SOMatic Viewer, an implementation of an interactive SOM visualization toolset in Processing and Java. The focus in this research will rely on the improvement of the implementation of Mr. Spöcklberger, as in his programmed library the computational processes to create a SOM take place.

The motivation for this project lies partially in a decrease of the computation time of the SOM computations through an improvement the SOMatic software components. The updated and extended Java library should then be integrated into a newly created Java cluster system library. This system is using the approach to spread the computations to the machines within this cluster, where all computations can be run in parallel. This addition in parallelization (SOMatic itself is already applying a parallelization on multiple cores on one machine) will help to further decrease the problem with computation time. Such an implementation would open the door for massive parallelization processes with very large SOMs or input data sets. Depending on the data size, the outsourcing of SOM computations to more powerful systems or facing very long computation times would then be obsolete.

The general objective of this project is to upgrade and extend a SOM implementation. In particular, its computation time should be decreased and the performance, as well as the usage of the output, increased. This improved SOM software will be tested with census data from Carinthia from 2001 and used on multi-temporal, multispectral image data. Within the GIS world, very large data sets exist that can benefit from parallel and distributed computing (Hawick, et al., 2003). This project aims to combine both these computer science technologies to develop an efficient speedup of the computation time for SOMs via SOMatic. The SOMatic Trainer 1.0 runs with Java 6 and is visualized with Processing 2. One aspect of improvement is to migrate the software to the latest versions to be applicable with Java 8 and Processing 3. This can already lead to an increase in compatibility and capability. A performance increase can also be achieved through the extension of the parallelization processes of the SOM computations. The first step consists of increasing the parallelization processes on one machine. The training process of SOMatic is already parallelized but the search of the BMU is not. This computation takes a lot of time, especially for a large numbers of input vectors. Therefore, the overall computation time can be decreased through a parallelization of the BMU search. Of high interest is also to be able to apply the parallelized BMU search after the training is finished. The integration of SOMatic to a cluster computing system would be the second step of parallelization. With the use of a distributed system library, the software will run on different machines that are connected within a cluster to enlarge the computation power. This will also help to increase the performance of SOMatic through a decrease of computation time for large input data.

SOMatic is used a lot by Dr. Skupin and other students at SDSU. So far, the output of this software is a Codebook (cod) file, which holds the trained SOM as well as an additional .sprj file, which contains information explaining the file paths to the in- and output data as well as the applied settings during the training. To be able to use the SOMatic output easier in Geoinformation programs or web applications, an output in the Geographic JavaScript Object Notation (GeoJSON) format would be very useful.

1.2 Problem Definition

In a time span of four years from the early implementation of SOMatic until this project, several newer programming versions were developed. Java 8 and Processing 3 are at the time when this project is conducted, the modern versions of these two programming languages. As one goal of this project is to expand the usage and applicability of the SOMatic Trainer, it is clear that the integration of newer software languages should have its place in this research. It is also easier to extend a software with more functionality written in latest languages as the compatibility with other software is usually just backward and not forward and the support for elderly versions will be terminated at some point after newer versions are public. Therefore, it is necessary to keep software up to date.

Another problem is situated in the topological alignment of the neurons within a SOM created by SOMatic. In some visualizations, there is a horizontal shift of the hexagonal neurons to the top, although it should always be to the right starting in the second row and moving upwards from the bottom left. This skewed alignment of the neurons is necessary because of their hexagonal form. The starting neuron on position (0/0) at the bottom left should always have two neighbors. In some visualizations, it has three neighbors. This is due to the fact that the origin is set to top-left and not bottom left. The visualization style with having this starting corner is not common within the geographic world. From a geographers' point of view, the origin should always lie in the first quadrant, meaning bottom-left. Furthermore, each second row starting from the bottom should be shifted to the right to fit a hexagonal structure and not any other row or column. The following Figure 1 shows the neurons with their coordinates within a different hexagonal alignment.

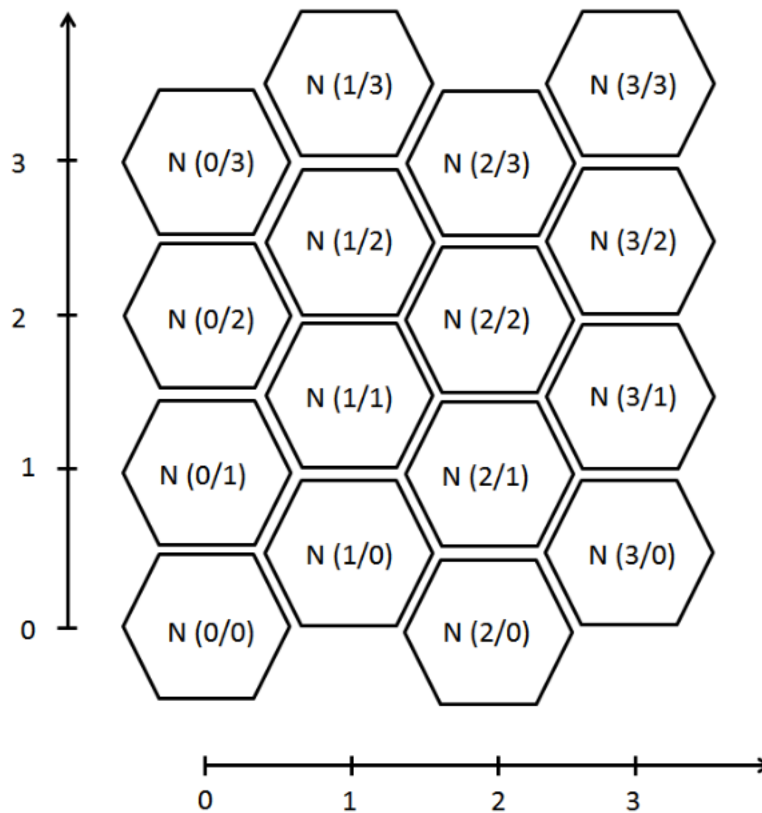

Figure 1 Hexagonal SOM display in coordinate system with different alignment (Spöcklberger, 2013)

The current output of SOMatic is a. cod file, which cannot directly be integrated into other applications like a web application for instance. This limits the software results in its usability within other programs. Additionally, the topological information is also not stored in the output files. Exporting the resulting SOM as well as the final BMU vectors in a more common and compatible format, which also supports geographic data would be of high value to increase the capabilities and utilities of SOMatic.

The computation time of a SOM depends on the number of input vectors, also called neurons, the number of training runs and the number of dimensions of the input vectors. Multispectral or hyperspectral data, for example, has a high number of input vectors with many attributes, which is referred to each pixel defined by its corresponding spectral bands. To generalize, the computation time is increasing with the amount of input data. SOMatic is already countering the time issue via a parallelized training of the SOMs. Thus far, there is no functionality in the SOMatic Trainer that allows projecting additional data onto an already created SOM. The whole process would have to be rerun with the old input plus the additional data. This would cost a lot of time, especially for a large input dataset with a high amount of dimensions. The BMU search after training, which counters that problem is not implemented yet. This step is embarrassingly parallel as no dependency between the BMU searches for the different input vectors is given.

Furthermore, the different threads do not have to communicate with each other in this finding step. Its sole purpose is to find the best suitable SOM neuron for each input vector via calculating the shortest distance of all the different attributes to it. A parallelized integration of this process within the software would lead to a significant decrease in the overall computation time, as not the whole process would have to be rerun when newer data of the same type should be visualized, but only an additional BMU search.

The possibility to distribute the computing processes in SOMatic not only on different threads but also on different machines within a cluster has not been touched as yet. Very large input datasets can also take a long time on parallelized local computations. Nevertheless, setting up and distributing the different computation processes in a distributed system takes time as well. It is important to find a threshold which defines what the lower margin in data size is to achieve a useful speedup when using SOMatic to compute SOMs distributed in a network cluster.

Depending on the applied parallelization technique, the quality of the output SOM can suffer from a partitioned computation. This problem is getting worse with an increase in the number of partitions and a decrease in the number of training runs performed within SOMatic. Especially a parallelization beyond one machine can raise the negative effect as the number of partitions could increase up to the number of available cores on all machines within the network cluster. To find an acceptable balance factor between quality and computation time will be crucial when running SOMatic in a parallel environment.

1.3 Research Questions

The following four research questions are planned to be answered within this project:

How can the output of SOMatic be more useful for further usage (in the GIS world)?

How can SOMatic be integrated into a cluster computing network?

What is the data size threshold beyond which it is faster to run SOMatic in a network cluster on different machines than on one machine?

What is the right balance factor between computation time and quality of the output SOM?

1.4 Methodology

The planned steps in the implementation phase of this project are as follows:

1. fixing and upgrading the status quo of SOMatic

Software version upgrades to run on the latest Java and Processing versions are useful to get a wider compatibility with other software and to improve the efficiency of the own software. A change of the used libraries in the Java code to version eight as well as the run with the integrated development environment (IDE) of Processing 3 will be applied in this step. Through the migration, some parts of the code might have to be modified or added depending on the version compatibility. The horizontal shift in the SOMs should always be to the right in each second row starting from bottom left. This issue will be tackled in the Java code where the SOM is created.

2. get SOM results in a geographic information software compatible format like GeoJSON

The two output GeoJSON files will represent the SOM after the final training stage and the BMUs for each input vector. This means they will be created after the finish of all computations within SOMatic. One GeoJSON object in the first file will represent one neuron and hold information about its 2D location as well as the trained data values. The GeoJSON objects in the second file will contain the BMUs. The format GeoJSON is predestinated in a way as it would be relatively easy to store each neuron as a GeoJSON object and is highly compatible with today's web applications when used Java or JavaScript.

3. extend the Software through implementing a parallelized computation of the best-matching unit (BMU)

The parallelized search for the BMU will be achieved through a data partition of the available input vectors, which can also be described as a divide and conquer approach. Each processor will find the BMUs in the SOM for its part of the input data.

4. extend the parallelization beyond one machine

The SOMatic computations will be distributed onto different machines connected in a network cluster. An existing library like Java Parallel Processing Framework (JPPF) will be used to achieve a distribution amongst several machines. JPPF is a software that enables programs with large processing power requirements to be run on different computers to reduce their processing time by a great amount. A master-slave structure will be applied, where two different approaches are possible. In the first one, each slave runs the whole process and sends its results to the master, which averages them.

In the second approach, the master distributes the initial SOM as well as equal-sized parts of the input vectors to each slave. The slaves send the master intermediate results, which are aggregated and redistributed to the slaves to ensure a higher quality. The end results will again be put together by the master.

5. validate the effects of parallelization on the speed and quality and apply SOMatic on data

It is important to identify the level of parallelization for a certain input data size. The quality of the results will also be evaluated via quality metrics like the average quantization error (AQE). The SOM algorithm will be used to apply a dimensionality reduction on data like census data and multispectral data.

1.5 Expected Results

The main result of this project will be an upgraded and extended SOMatic Java library. It will use the latest version of Java and run with the latest version of Processing. The new SOMatic output will also be available as a GeoJSON file. GeoJSON is a wide spread geographic file format, which can easily be integrated into web applications. This will assure that the benefits of using SOMatic over other SOM implementations will yield increased functionality by a significant amount. Through extending the parallelization on a local and distributed basis, the computation time for SOMs is further decreasing. The search of the BMU after training will be parallelized locally.

The search of the BMU can be described as the second primary process in the SOM algorithm besides the training. Its computation on multiple cores will decrease the overall computation time greatly. Within a distributed systems framework like JPPF, a new Java library will be created where SOMatic will be used to apply a cluster computation on different machines connected within a network. It will use a master-slave architecture to control and distribute the computation tasks. This newly created library should also be portable to any other network cluster and should easily be adaptable to other parallelizable software processes. It should be platform independent as well. Its only requirements will be Java, JPPF and a connected network cluster. The achieved speedups, as well as quality decreases, will be visualized in diagrams and evaluated with quality metrics.

It is of high importance, that the time and quality decrease are balanced. The user will also be able to directly tackle this topic as it should be possible for her or him to choose the level of parallelization and with this way control the speed increase versus quality decrease.

1.6 Thesis Structure

This thesis gives in the first chapter an introduction to the topic and research field of the project. Following on that are the relevant literature and best practice examples in the chapters two and three, a workflow in chapter four, the explanation of the implementation processes in chapter five and results of the extended software in chapter six. This report is rounded up with a discussion and conclusion of the methods and results in chapter seven as well as possible expansions of the software for the future in the chapter eight.

2. Theoretical Background

This chapter describes the topics, which were tackled in a theoretical approach to deepen the knowledge in them. It starts with the SOM and the SOM algorithm, where the main computation steps, as well as the possibility for parallelization, are explained. After that, the important languages are explained, before getting to the end of this section, where different distributed system libraries and methods are explained.

2.1 Self-Organizing Map

The SOM was developed by Professor Toivo Kohonen in 1981 and is a useful tool to visualize high-dimensional data (Kohonen, 1998). It is an Artificial Neural Network (ANN), which is based on unsupervised machine learning.

The difference to supervised learning lies in the unknown information about the output. The input data is described via neurons usually displayed in a two-dimensional grid that can also be seen as a map. This feature map can be helpful to analyze and detect features within the input space (Lawrence, et al., 1999). Therefore, the SOM can be used to identify clusters of the input data as it provides a visual representation of similar data instances (Wittek, et al., 2016).

The most common neuron structures of a SOM are either rectangular or hexagonal, where the second one provides a higher quality (isotropy) in visualization (Kohonen, 1998). The following figure gives an example of the application possibility of the SOM. It shows a part of the SOM that was trained with data contained in over two million medical publications (Skupin, et al., 2013). The whole SOM consists out of 75,000 neurons and has 2300 dimensions, which represent the top ten percent of the most frequent terms used in the medical publications.

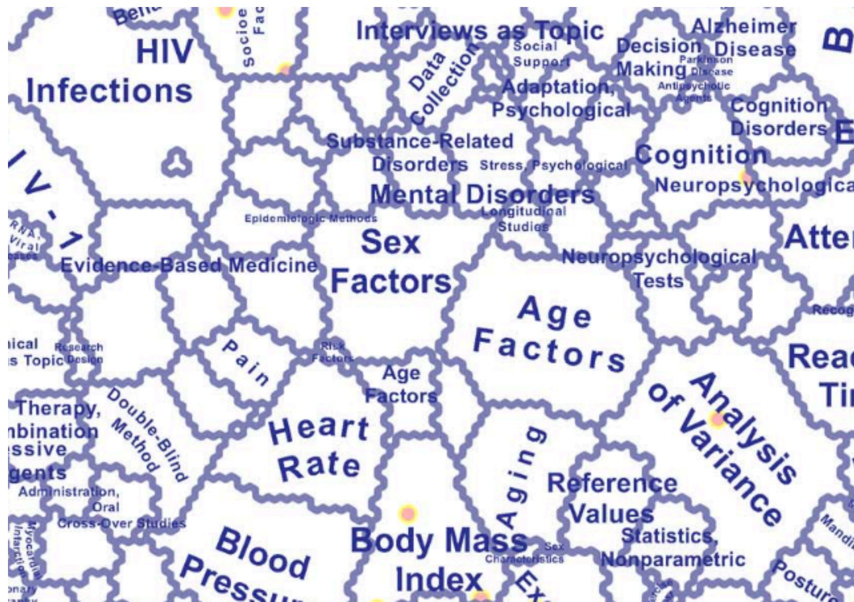

Figure 2 SOM created out of words in medical publications (Skupin, et al., 2013)

2.2 SOM Algorithm

Kohonen, the inventor of the SOM, describes the central idea of the SOM algorithm is that every input data item selects the best matching neuron to itself. This neuron, as well as a part of its spatial neighbors, will then be modified for a better matching (Kohonen, 2013). Figure 3 shows the sequence of the SOM algorithm for one training epoch. The following sections describe two different types of the SOM algorithm using formulas derived from Lawrence, et al (1999).

To explain the steps mathematically, we assume a set of input vectors **x** and assign a weight vector $\mathbf{w}_k$, $k = 1,...,K$ to each of the K nodes randomly within the initialized SOM. A time index t is introduced, which also stands for the training run. This means $x(t)$ is an input vector used in the algorithm at time t and $\mathbf{w}_k(t)$ is the associated neuron weight vector at time t. The initial values for the neuron weights are assigned randomly.

2.2.1 Conventional SOM Algorithm

In this version of the SOM algorithm, the weights of the neurons are updated recursively after the use of one input vector (Lawrence, et al., 1999). After the selection of one input vector the distance from this vector, to all neurons in the SOM is computed. The neuron with the smallest distance to the input vector is chosen as the BMU. Its weights, as well as those of the neighboring neurons, are updated so that they become more similar to the input vector. Figure 3 shows this linear procedure, which represents one training iteration of the SOM algorithm.

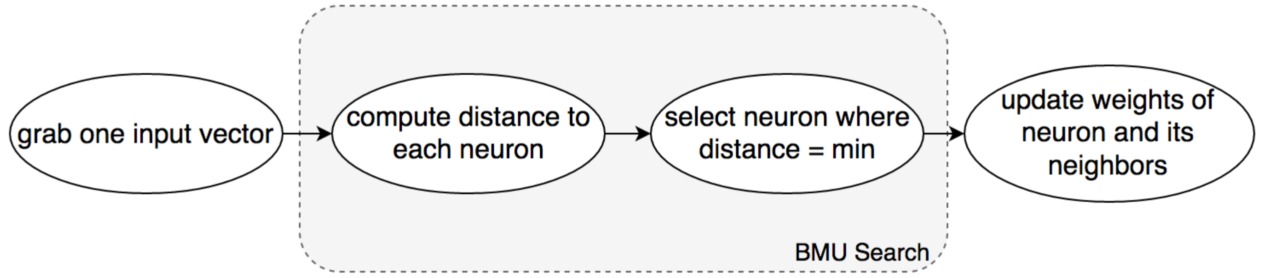

Figure 3 Explanation of one training iteration of the SOM algorithm

To compute the distance from the input vector to the neurons the Euclidian, Cosine or Manhattan distance functions are commonly used and also available within the SOMatic trainer (Spöcklberger, 2013). The following formula is using the Euclidian distance.

(1)

$$d_k(t) = \|\mathbf{x}(t) - \mathbf{w}_k(t)\|^2$$

Subsequently, the minimum out of these distances is selected. The neuron that is associated with this distance is called the BMU of this input vector at this training iteration. Together with the distance computations, these two steps make up the BMU search. The distance to the BMU is here denoted with the subscript c.

(2)

$$d_c(t) \equiv \min_k d_k(t)$$

The weights of this neuron, as well as those of its neighbors, are updated with the formula

(3)

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \alpha(t) h_{\mathrm{ck}}(t)[\mathbf{x}(t) - \mathbf{w}_k(t)]$$

where $\alpha(t)$ is referred to the learning-rate factor and $h_{ck}(t)$ is the neighborhood function. The learning-rate factor is regulating the amount of changes in the weight vectors and is reduced in a monotone way as the training progresses. The neighborhood function defines the magnitude of change of $\mathbf{w}_k(t)$ to an input vector, which is most closely similar to $\mathbf{w}_c(t)$. It is usually a decreasing function with respect to the increase in distance from node c to k. The standard Gaussian neighborhood function is

(4)

$$h_{ck}(t) = \exp(-\|\mathbf{r}_k - \mathbf{r}_c\|^2/\sigma(t)^2)$$

where $\mathbf{r}_k$ and $\mathbf{r}_c$ are referred to the coordinates of the node *k* and the current BMU *c*. The width of the neighborhood function is described by σ(t) and also decreases as the training process goes on, usually from a distance of one dimension of the 2D plane down to the size of a neuron.The explained formulas are used in Figure 4 and explain the conventional SOM algorithm with a pseudo-code.

```
initialize weight vectors
t = 0
for (epoch = 1, ..., N_epochs)
    interpolate new values for α(t) and σ(t)
    for (record = 1, ..., N_records)
        t = t + 1
        for (k = 1, ..., K)
            compute distances d_k using Eq. (1)
        end for
        compute winning node c using Eq. (2)
        for (k = 1, ..., K)
            update weight vectors w_k using Eq. (3)
        end for
    end for
end for
```

Figure 4 Pseudo-code representing the conventional SOM algorithm (Lawrence, et al., 1999)

2.2.2 Batch SOM Algorithm

The difference to the conventional SOM algorithm lies in the updating of the weights, which are updated after one epoch, also called one batch. One epoch is reached after the whole input data, respectively the input vectors were used one time. This means that here, the BMUs for all input vectors are searched and afterward its weights and those of its respective neighbors are updated. The BMU search is an embarrassingly parallel process, meaning that each search is completely independent of the others and no communication has to take place between the processors working on this task. The information, to which input vector one neuron is the BMU has to be stored in a list, but this operation does not involve any computation. Hence, the batch is faster than the conventional version as it has less computational effort by only making an update of the weights once per epoch compared to once per BMU search. Furthermore, the batch version does not require any specifications of the learning rate factor (Kohonen, 1998).

The following formulas are obtained from the same source as those in the prior section (Lawrence, et al., 1999). The weights are updated with the formula

(5)

$$\mathbf{w}_k(t_f) = \frac{\sum_{t'=t_0}^{t'=t_f} \tilde{h}_{\mathrm{ck}}(t')\mathbf{x}(t')}{\sum_{t'=t_0}^{t'=t_f} \tilde{h}_{\mathrm{ck}}(t')}$$

The result $\mathbf{w}_k(t_f)$ are the mentioned weights, t_0 and t_f mark the start and finish of the current epoch. To compute the winning node the formulas

(6)

$$\tilde{d}_k(t) = \|\mathbf{x}(t) - \mathbf{w}_k(t_0)\|^2$$

(7)

$$d_c(t) \equiv \min_k \tilde{d}_k(t)$$

are used. $\mathbf{w}_k(t_0)$ refers to the weight vectors that were computed in the final step of the previous epoch. The in (4) described neighborhood functions are using the winning nodes from (7). The width of the neighborhood function is also decreasing monotonically throughout the training phase as it is in the conventional algorithm. The following Figure 5 represents these formulas in a pseudo-code example.

```
initialize weight vectors
t = 0
for (epoch = 1, ..., N_epochs)
    interpolate new value for σ(t)
    initialize numerator and denominator in Eq. (5) to 0
    for (record = 1, ..., N_records)
        t = t + 1
        for (k = 1, ..., K)
            compute distances d̃_k using Eq. (6)
        end for
        compute winning node c using Eq. (7)
        for (k = 1, ..., K)
            accumulate numerator and denominator in Eq. (5)
        end for
    end for
    for (k = 1, ..., K)
        update weight vectors w_k using Eq. (5)
    end for
end for
```

Figure 5 Pseudo-code representation of the batch SOM algorithm (Lawrence, et al., 1999)

One drawback of the batch computation is the failure of the good organization of code-vectors, which are distributed in a special shape like the one of a "W" (Fort, et al., 2002). Figure 6 and Figure 7 show this behavior in comparison to the result of the conventional on-line SOM algorithm. Both computations were using the same initial state of the neurons.

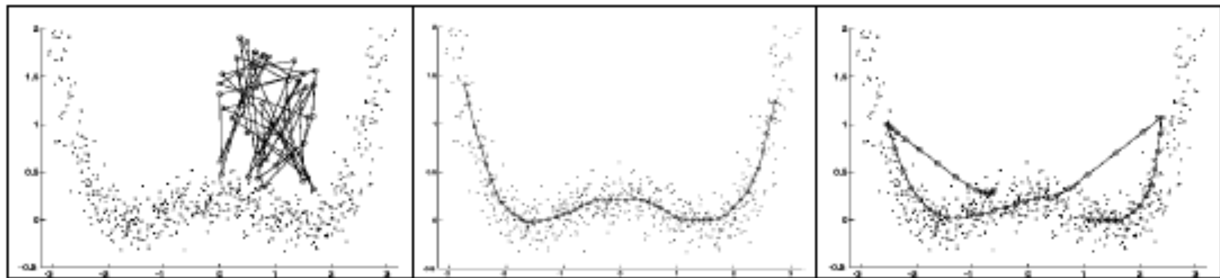

Figure 6 Starting state (left) and 2D comparison of the results of the on-line (middle) and the batch computation (right) (Fort, et al., 2002)

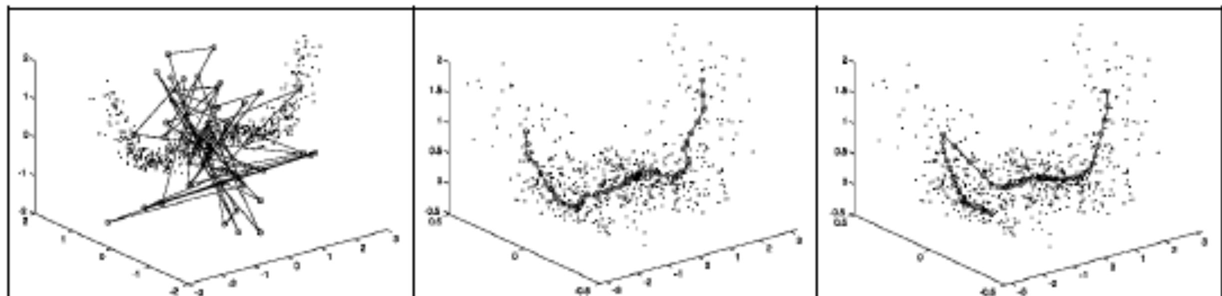

Figure 7 Starting state (left) and 3D comparison of the results of the on-line (middle) and the batch computation (right) (Fort, et al., 2002)

2.2.3 Parallelizing the SOM Algorithm

In general, there are two different approaches to apply a parallelization: data or network partitioning (Lawrence, et al., 1999). As the name tells it already the input data is split up in the data partition method onto different threads or machines. It is of high importance to make the partitions of an equal size to prevent latency. So each processor should work with the same amount of data. The network, in this case, the neurons, needs to be available to all threads either as a copy or through accumulators that are pending updates to the map (Skupin, et al., 2013). After the individual weight updates of the neurons of each processor, the outputs have to be merged together and redistributed in order to maintain an up-to-date SOM and be able to continue with the training. Therefore, the batch version of the SOM algorithm is more suitable for the data partition approach as the updates are only computed after one epoch. This reduces the overall computation time as the merge and redistribution of the updated neurons cost communication time.

Depending on the frequency of the merge (if not using the batch-version), the result of this method can be different compared to a linear implementation. The advantage of the data partitioning is its effectiveness on a large amount of input data like in Skupin et. al (2013). Some other examples of this method are also described in these two theses (Silva & Marques, 2007) (Yang & Ahuja, 1999).

In the network partitioning method, the neurons are split up. So each processor is just using and applying computations on a defined part of the SOM. Therefore, the input data has to be accessible by or copied to all processors. In one version of this method (Lawrence, et al., 1999), each processor makes the BMU search for the same input vector for its set of neurons. The overall BMU is then found by comparing the BMUs of each processor and picking the neuron with the minimum distance to the input vector out of them. After that, each processor is then updating its affected neurons. An advantage of this method is that it produces an exact agreement (within a round-off error) with the conventional algorithm (Lawrence, et al., 1999). The downside, however, is that the processing of each input vector produces a latency through communication and also updating of the weight vectors. This results usually in a longer computation time compared to the data partitioning method. The third form of parallelization, dividing the training runs onto different processors, is applied in SOMatic (Spöcklberger, 2013) and explained in section 3.3.1 in this thesis.

2.4 Important Languages

2.4.1 Java

Java is an object oriented, concurrent and class-based programming language, which is one of the most used programming languages worldwide. It was originally developed by the Sun Microsystems, which has been acquired by Oracle (Wikimedia Foundation Inc., 2017). The syntax of this programming language has its origin mostly in the languages C and C++. The latest version of Java is Java 8. The following Figure 8 shows a basic “Hello World” example written in Java.

```
public class HelloWorldExample {
    public static void main(String[] args) {
        // Prints "Hello World." onto the console
        System.out.println("Hello World.");
    }
}
```

Figure 8 Java "Hello World" example code

The filename in Java must be equal to the public class name it contains. So "HelloWorldExample.java" has to be the name of the file that contains this code. Before the code can be run it has to be compiled. Compiling is the transitioning of the code from a human-readable language to a machine-readable language. After this step, a new file appears, which has the ending ".class" and contains the information of the ".java" file in a machine-understandable language. As shown in the example, the main method is used in Java to obtain a result of the code and display it for example on the console. The comments can be written after two forward slashes as shown in the example, or also between a forward slash plus an asterisk.

There exist several keywords in Java and some of them are shown in the code snippet above written in a wine-red color: public, class, static and void. The keyword public means, that the following method or class or variable can be accessed from outside of the method or class where it is located. The term for this keyword is access level modifier. There are also other access level modifiers: protected and private. These two limit the access to inherited classes (protected) or only the class or method where the variable or method is located (private). The next keyword "class" refers to the first level in the Java code. One .java file can contain several classes, but only one public class, which must have the same name as the file.

A method that is called "static" cannot access other class members, which are not static. Furthermore, it is not possible to create instances from this method. This means, that there can only exist one version, which is associated with the class it is located in. The last keyword "void" is referred to the return value of a method. When a method is written as "void", it means that it does not return any value. The main method, for example, has to be a void method. As the library of SOMatic (Spöcklberger, 2013) is written in Java, there was no other language choice possible to choose in this project. This programming language is the main language besides python that is taught at CUAS in Villach and also used a lot in San Diego, which is one important reason why it is used.

2.4.2 GeoJSON

GeoJSON is a data format used for the encoding of different geographic data structures using JavaScript Object Notation (JSON) (Butler, et al., 2016). It was first introduced in 2008 and differs from other GIS standards in a way that it was written and maintained by a working group of developers and not by a formal standards organization (Wikimedia Foundation Inc., 2017). GeoJSON supports seven different geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection, which refer to the definitions of the Open Geospatial Consortium (OGC) (1999). Within Figure 9 one can see an example of how GeoJSON looks like. It shows a "FeatureCollection" that contains three features having different geometries.

```
{ "type": "FeatureCollection",
    "features": [
      { "type": "Feature",
        "geometry": {"type": "Point", "coordinates": [15.5, 15.5]},
        "properties": {"definition": "city"}
        },
      { "type": "Feature",
        "geometry": {
          "type": "LineString",
          "coordinates": [
            [2, 2], [2, 4], [4, 4], [4, 6]
            ]
          },
        "properties": {
          "definition": "river",
          "length": 6
          }
        }
      ]
    }
```

Figure 9 GeoJSON example

Within the literature research for this project, no SOM implementation could be found that exports the output in the GeoJSON format. A total of two GeoJSON files will be created as an extension to SOMatic. The first one holding the finalized neurons of the SOM after the training and the second one holding the BMUs also after the final training. The neurons will be stored as features of the type polygon and having their attribute values in the "properties" of the corresponding GeoJSON object.

2.5 Distributed Systems

2.5.1 OpenMPI

The Open Message Passing Interface (MPI) project is developed and maintained as an open source project by a consortium of academic, research and industry partners (Open MPI Project Team, 2017). It combines the technologies from four other MPI projects and aims to use them to create a world class open-source MPI implementation (Wikimedia Foundation Inc., 2017). An MPI can be used to setup a communication between different parallel computing sources. Although it was used in one best-practice example (Wittek & Darányi, 2012) that is also explained in section 3.2.2, this technology is seen as not applicable for this project and time scope as it only has libraries written in C, C++, or Fortran and would, therefore, require more time than available.

2.5.2 Apache Hadoop

The Apache Hadoop project software is used within distributed computing to process large data sets across computers in a cluster using simple programming models (Apache Foundation, 2017). It is designed to detect and handle failures of nodes in the cluster and has, therefore, a high availability. Hadoop uses the MapReduce programming model, which is an associated implementation to handle big data sets with a parallel, distributed algorithm within a cluster (Dean & Ghemawat, 2004). The "map" is a function specified by the user that generates out of one key/value pair a set of key/value pairs and the "reduce" function merges all intermediate values, which are associated with the same intermediate key (Dean & Ghemawat, 2004).

Figure 10 shows an implementation of the MapReduce programming model that was used by Google. The first step is to split (also called fork) the input data into *M* pieces. One of the two main components (called copies in MapReduce), the master, is assigning *M* map tasks and *R* reduce tasks to the second main component called workers. These workers need to be idle to get a task assigned to them. Those workers who got a map task assigned to them read the content of the corresponding input split and parses key/value pairs to the user-defined map function. The map function produces also intermediate key/value pairs, which are buffered in memory. These buffered pairs are periodically forked into *R* regions and written to the disk by the partitioning function. The master gets the location of these pairs and forwards them to the reduce workers. The reduce workers read the data and sort it depending on the keys as data with the same key will be grouped together.

For each unique key, the reduce worker passes the key as well as the corresponding set of intermediate values to the reduce function that has been defined by the user. The output of this reduction is then written to an output file. After all the map and reduce tasks have been completed, the MapReduce call in the program of the user returns back to the user code (Dean & Ghemawat, 2004).

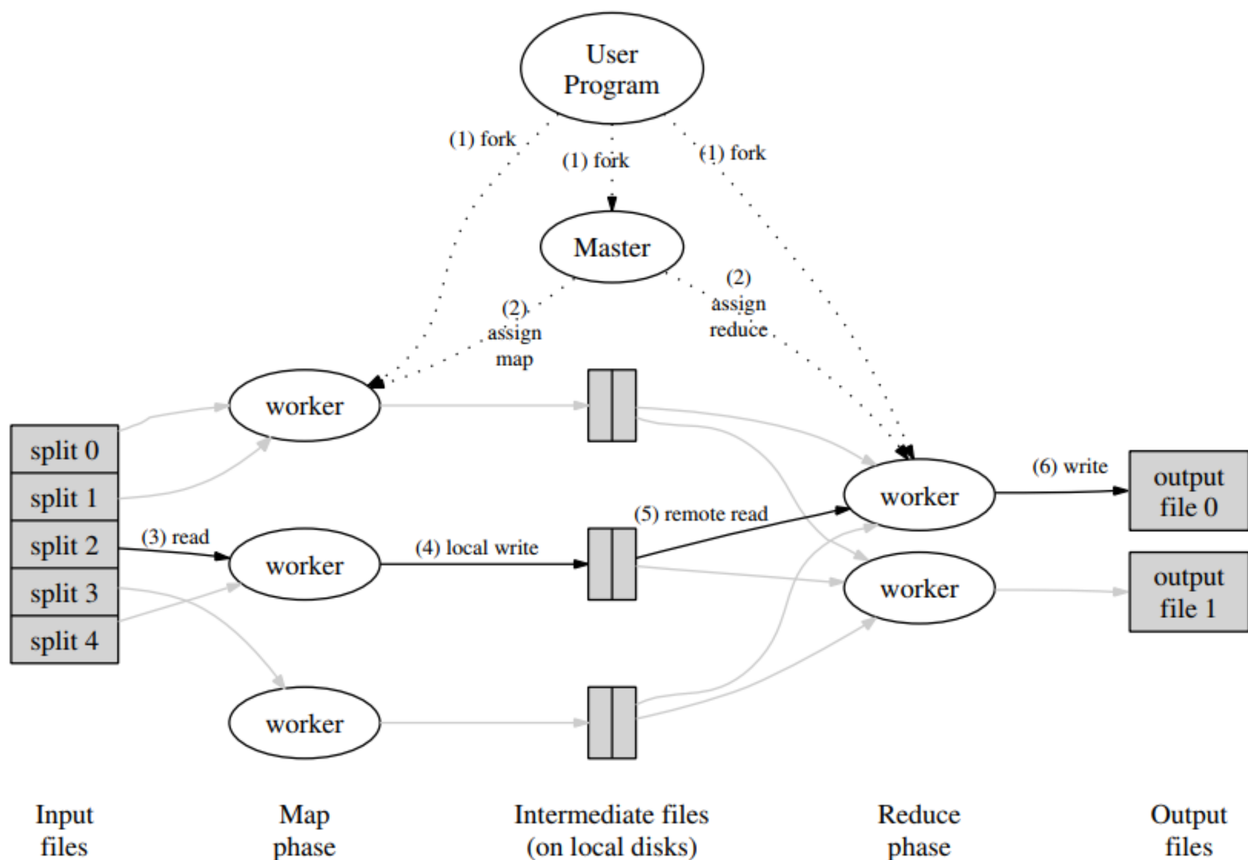

Figure 10 Example of the flow of a MapReduce operation (Dean & Ghemawat, 2004)

Apache Hadoop is also written in Java and would, therefore, fit this project looking at it from this perspective. However, it has been noted that the MapReduce queries usually take several minutes or more and are best used in offline-modus without a human sitting in front of the PC (White, 2015).

2.5.3 Java Parallel Processing Framework

JPPF is a Java library that is used to run applications on any number of computers. It splits the task into smaller parts, which can be executed in parallel. It works on any operation system (OS) that supports Java (JPPF.org, 2016). The network topology consists of three different components: client(s), server(s) and nodes. The following Figure 11 gives a possible setup of this topology.

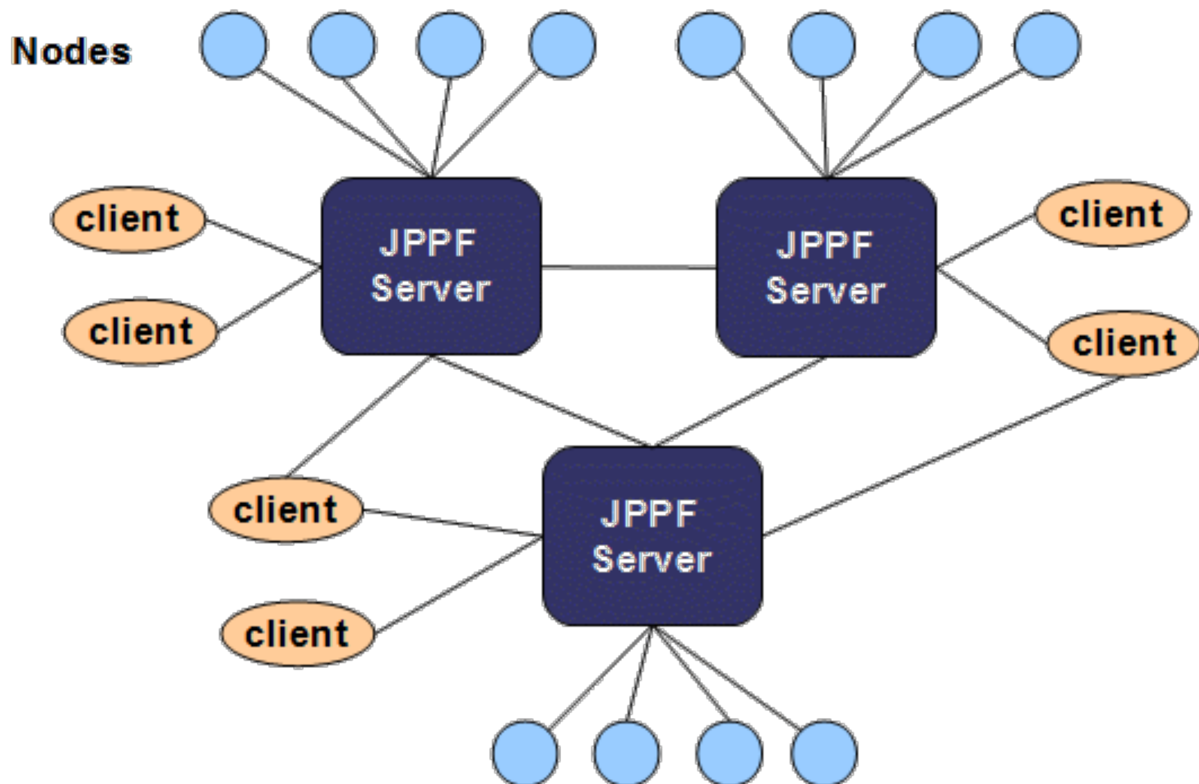

Figure 11 Overview of a JPPF network topology (JPPF.org, 2016)

One network can consist of several servers, clients and nodes. The clients submit the work to a server. There, it enters a job queue and the server filters on the available nodes. Via a load balancing mechanism, the tasks are distributed to the nodes, which perform the task and send the result back to the server. The last step is the sending of the result from the server back to each client. Figure 12 shows this stepwise workflow within the JPPF topology.

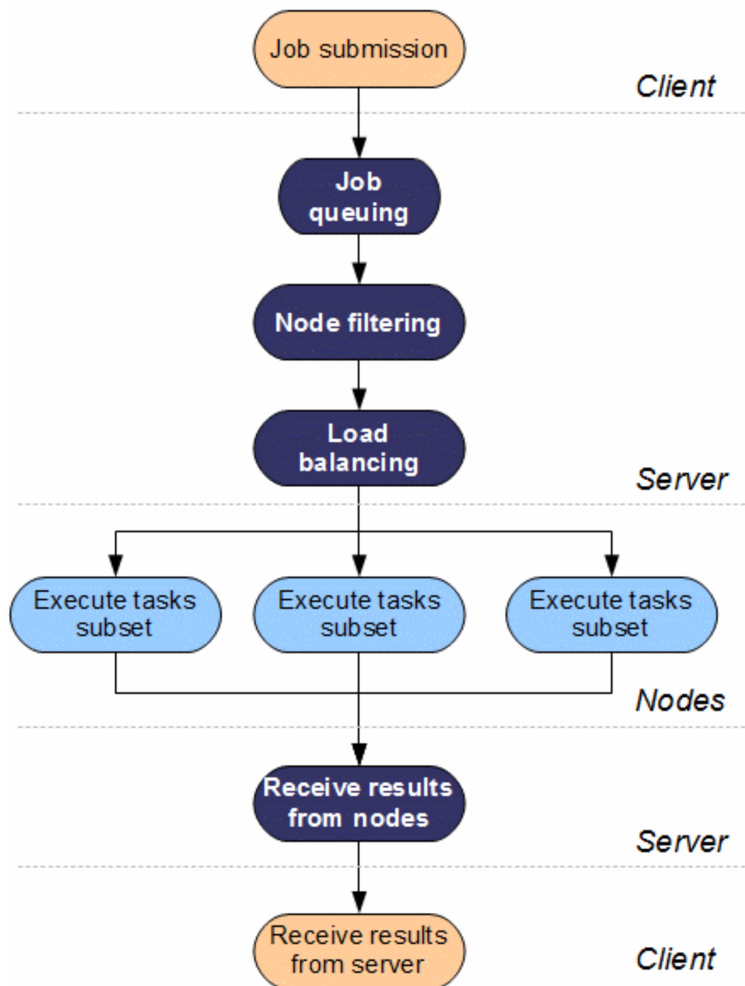

Figure 12 Job-Workflow in the JPPF topology (JPPF.org, 2016)

Parallelism occurs within the JPPF network topology at all three components. On the client-side, a configured pool of connections allows the sending of multiple concurrent jobs to one server at the same time. The size of this pool determines the number of jobs that can be sent. Through multiple clients, another form of parallelism is given as they can also send their workload to a server at the same time. The third appearance of parallelism on the client-side takes place if a mixed local and remote computation are enabled. JPPF offers the possibility to execute a job on a node and on the client at the same time to increase the computing power. On the server-side, parallelism can be achieved through a connection to more than one client and/or more than one node. The server has an important role in the network as it regulates the workload to each node with a load balancing mechanism. This assures, that the amount of the work to compute is distributed equally to all nodes in the cluster. Another mechanism called Service Level Agreement (SLA) can be defined by the user and tells the server to which node a particular job should not be distributed.

This can be useful if the cluster is heterogeneous considering the nodes and the user has concerns about performing particular jobs on some nodes. The user can also define the size of the network thread pool, which defines with how many nodes one server can communicate in parallel. If a system consists of more than one server, then one of them sees the other one as a client if the connection is one-directional. If it is bidirectional, then both servers can act as server and client depending on who sends a job to whom (JPPF.org, 2016). The node-side exploits parallel processing via using a thread pool to compute the received tasks. The size of this pool accounts the actual number of available processors on a specific node.

As JPPF is written in Java and seems to be an easy to understand and use tool to setup and run distributed computations in a cluster it is seen as most suitable for a usage within this project. One does also not have to think about the details considering message passing and load balancing. This is implemented and applied automatically in JPPF. Furthermore, it looks applicable to integrate SOMatic into a JPPF library to be able to use this SOM implementation within a network cluster.

3. Practical Background

This chapter is divided into three sections explaining different best-practice examples: parallel SOM implementations on a local machine in 3.1, distributed SOM implementations onto different machines within a network cluster in 3.2 and the SOMatic software in 3.3.

3.1 Parallel SOM Implementations

Six different parallel implementations of the SOM algorithm were chosen as an important literature basis of this thesis and are described within the following pages.

3.1.1 Hybrid Parallel SOM Algorithm

The method to apply the SOM algorithm explained here focuses on a mixture of the two common approaches to gain a speedup in computation time of a SOM: data-partitioning and network-partitioning (Silva & Marques, 2007). The batch data-partitioning algorithm is used at the beginning to obtain a first ordering of the topology of the map. After this initial global ordering, the BMUs are distributed in an adequate way globally over the map.

Via calculating an input data histogram, the map is segmented into different parts, which is referred to the network-partitioning method. The histogram is needed to be sure that the partitioning is executed in a balanced way. The number of segmentations is equal to the number of the available processing nodes. Each node is then focusing only on its part of the partition until a new segmentation is applied.

The reason behind this method is the fact, that the position of the BMU for each input pattern is generally situated in a small area of the map and therefore situated in the region allocated to one of the nodes. If the winning neuron for an input pattern is located in a contiguous region, it can lead to some errors. These errors are therefore weakened due to the following segmentation stage that repeats the process again. As the size of the number of neighbors which are considered is decreasing by time, the frequency of updates in these bordering regions is going down as well. It was tested with a pre-processed dataset, which resulted in 106 attributes for 6054 randomly chosen input vectors. Figure 13 shows the execution times of this method for different neuron sizes compared with the times of the classical Batch data-partition method (Kohonen, 1993).

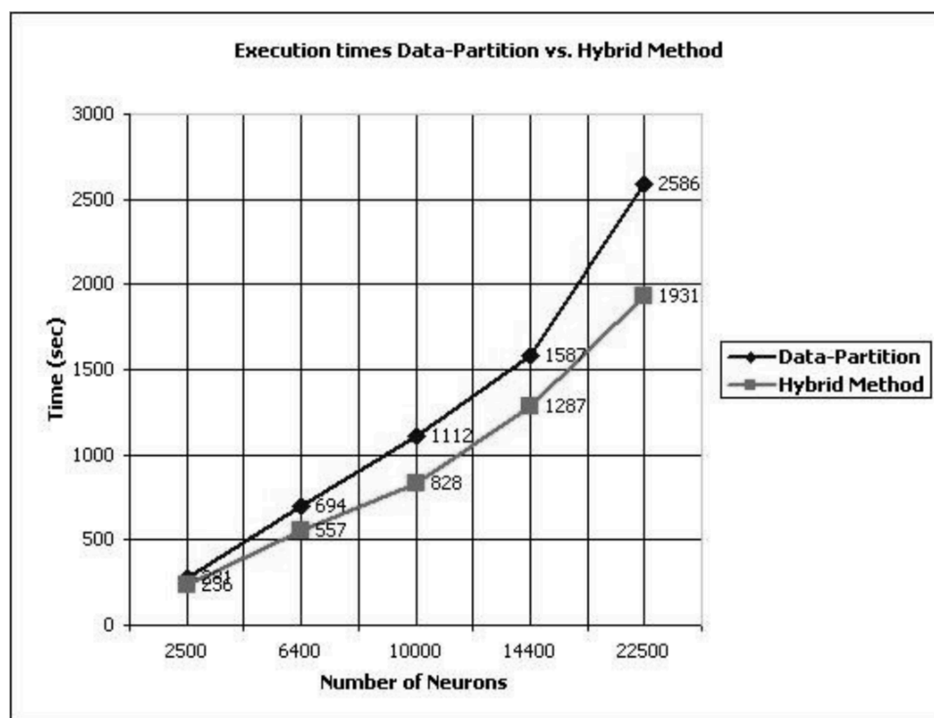

Figure 13 Execution times of data-partition vs. hybrid method (Silva & Marques, 2007)

The results of this method cannot be equal to the results of a normal data-partition approach, but the similarity level is very high. Additionally, the obtained information out of these outcomes is preserved as well. In view of the fact that the average speed-up of this method compared to the batch data-partition method is 1.27, its usage for large maps whilst accepting some small discrepancy in the results can be confirmed.

3.1.2 Speculative Parallel SOM Algorithm

In this paper, the members propose a new method of a parallel approach of the SOM algorithm, which outperforms classic map-partitioning implementations (Garcia, et al., 2006). They used simultaneous multithreading (SMT) processors, which opens the possibility to work on several instructions at the same time. This increases the efficiency of the processors computing powers because the operational latencies are hidden. These processors can be described as a set of logical processors, which split some resources among them. The code used in this project comes from the SOM-PAK (Kohonen, et al., 1996) implementation originally coded at the Helsinki University of Technology. This alternative implementation (Garcia, et al., 2006) uses the assumption, that it is possible to train two consecutive winning nodes in parallel if their neighborhoods are not overlapping. The finding of the BMU is always performed in parallel and the training of their respective neighborhoods is dependent on a possible sharing of the same nodes. If they do not overlap, the master, as well as the slave of the architecture, can compute the training process, but if they do overlap, the slave has to wait until the master has finished its updating. An overlap also means that the speculation has been wrong. After the update, the master and the slave switch their roles. This master-slave architecture can be seen in Figure 14.

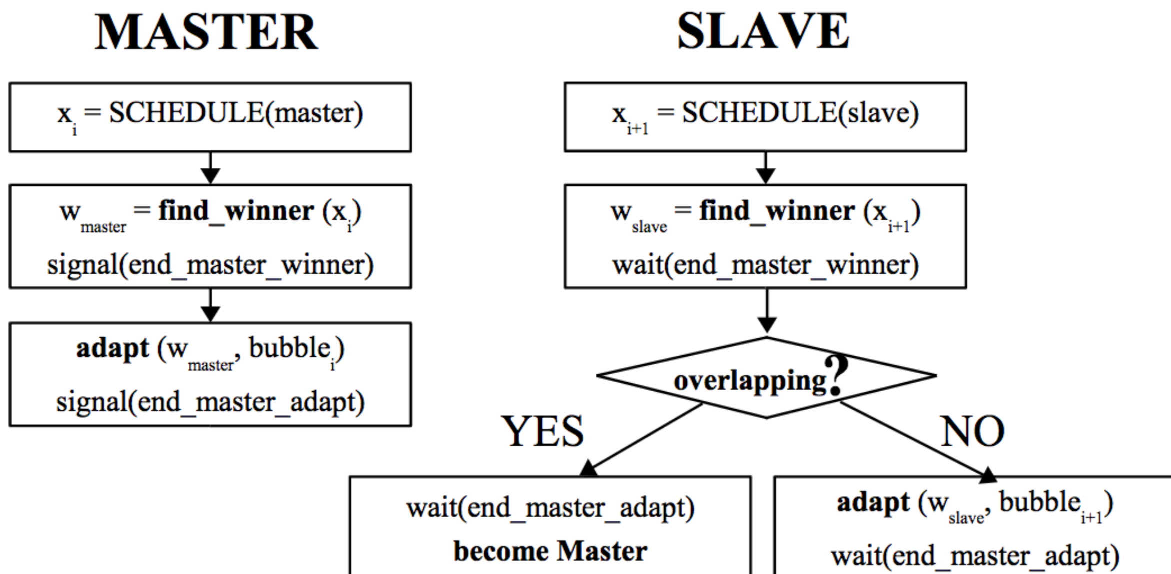

Figure 14 Master-slave architecture in a speculative parallel SOM algorithm (Garcia, et al., 2006)

As the chances of overlapping are decreasing during training due to the decrease of the size of the radius, which determines how many nodes are updated around the BMU, the speculation starts after ten percent of the iterations have been finished. In the initial warm-up phase, a map-partitioning approach is applied on the threads in parallel. The disadvantage here is that with a wrong speculation some resources were used to useless computations, which have to be re-executed. The advantage is an average increase in the speedup of 5 compared to the original SOM-PAK code.

3.1.3 Data Partition Method for Parallel SOMs

To apply a data partition approach is another possibility to achieve a parallelizable structure (Yang & Ahuja, 1999). This implementation achieves an average speed up of 3.15 by partitioning the training vectors into clusters, which are then assigned to processors for computing. It is based on the observation of the dynamic behavior and the characteristics of the neighborhood function in the SOM algorithm by Kohonen (Kohonen, et al., 1996).

The algorithm is described as follows (Yang & Ahuja, 1999): N is the number of processors, I the number of input vectors and M^2 the number of nodes in the map. The input vectors should be clustered into N clusters so that each processor can use one of these clusters as input data and the maps can be formed concurrently. Each processor should have about 1/N input vectors. This approach can be considered as a divide-and-conquer method as it decomposes the original problem into a set of subproblems which can be solved at the same time. The aim of this method can be described as dividing the original problem into N sub-problems such that they can be processed in parallel. A disturbance of this parallelization only arises if the neighboring nodes of one winning node are not within the same sub-map. The downside here is that the processors need to communicate with each other because of this possibility. This process makes sure that the overall consistency is still given. One processor informs in such a case the others to update the weights of their nodes as the neighboring function of the winning node overlaps with the nodes, which are in one or more other processors. This project tries to divide the problem in such a way that the occurrences of these overlapping cases are rare.

This data partition algorithm tries to make a good usage of the two training phases of the SOM. In the first phase, a coarse training takes place where the reference vectors are roughly ordered. The second phase is then the fine tuning of the values of the reference vectors. Furthermore, the neighborhood radius decreases from the first to the second phase but the computation time increases a lot. It is more than ten times longer than the first phase. The data is partitioned after the first phase as the mapping of the input vectors to the nodes is rather stabilized at that point and the neighborhood function (the radius) is smaller in the second phase.

In the testing of this implementation, the task was to solve an image coding problem via mapping 2048 training vectors into an 8x8 SOM using four processors. Their results show that the probability of the need to synchronize the different weight matrices in the sub-maps lies at roughly five percent. The achieved speedup can be numbered at 3.15 compared to running the algorithm on a single processor.

Figure 15 a comparison between the original image, the result of applying the conventional SOM and the result with the parallel SOM algorithm. This proves that the increase in speedup does not trade with a decrease in quality.

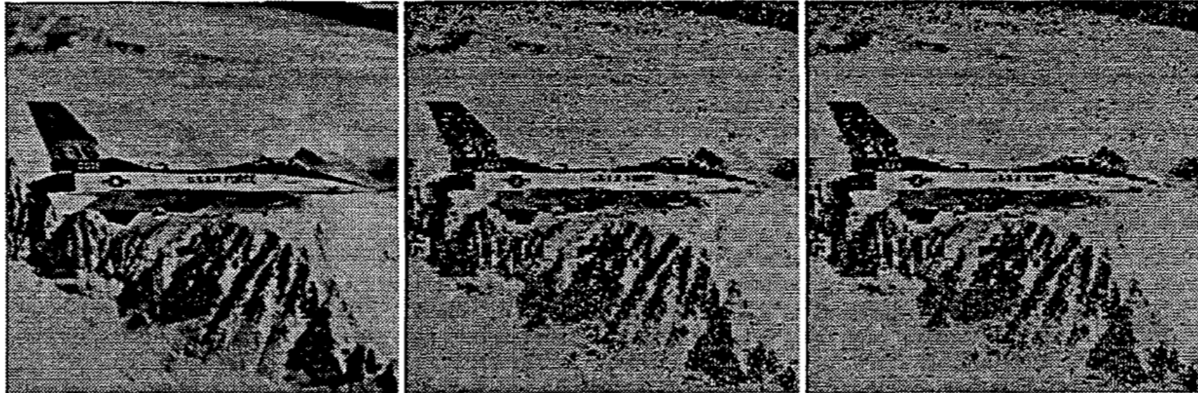

Figure 15 Comparison of raw image, conventional SOM and parallel SOM (Yang & Ahuja, 1999)

3.2 Network-Distributed SOM Implementations

This chapter gives two examples of a cluster computation of the SOM algorithms, where connected machines within a network were used. The second example also explains how SOMs can also be computed with the help of graphic processing units (GPUs).

3.2.1 Training a SOM distributed on a PVM network

A distribution of the computations within a SOM to several PCs using a Parallel Virtual Machine (PVM) network is described by Bandeira, et al. (1998). PVM enables to use connected computers that can have different operation systems like UNIX or Windows as a single UNIX machine via using the programming language C for instance. In this project they used PCs of the computer laboratories in their University to train the SOM networks. Given a number of processors N_p, a coordinator process within N_p called C, the number N_t of each training pattern x_i and neurons that form a SOM N_n, the approach of this implementation is described in five steps (Bandeira, et al., 1998):

1) Assigning of the neurons: The N_n neurons are assigned to the processors in a way that each one of them gets approximately the same number of neurons (N_p / N_n). Furthermore, they should be evenly distributed over the processors for any given area of the SOM.
2) Setup of the training set: All of the N_t patterns are send together with the initial training parameters to each processor.
3) Training of the neurons: For each x_i in the training set, 3 phases are computed. The first one is called calculation phase. Here, the local winner neurons are calculated. This phase runs in parallel as each processor calculates that for its neurons. The second phase is the voting phase, where the processors send the coordinates their winner neuron to the coordinator process C. This coordinator selects the global winner and distributes its location back to the processors. In the third phase each processor updates the neurons according to the update function (update phase). Additionally, the training parameters are also updated.
4) Repetition of step 3 until a stopping criteria is met.
5) Sending of all neurons back to the coordinator.

Within this experiment a network with up to 12 PCs was used. The possible speedup was highly depending on the processing load that was required before each synchronization in the update phase. Therefore, very large pattern vectors with 1024 features were used and the number of neurons on the map was varied. The size of the maps varied from 5x5, 10x10, 20x20 and 40x40 squares, where the numbers refer to the neurons meaning a 5x5 square consists of 25 neurons. The decrease of the radius per training run was set at one unit, which means that a map with 5x5 neurons would have 5xN_t iterations on the patterns and the biggest one would have 40xN_t runs. Therefore, the number of training patterns was decreased with an increase in map size in order to counter the rise of computation time that would occur within that radius decrease. The numbers of training patterns were chosen in a way that each map would require four times more calculations than the previous one had.

Figure 16 shows the curves computation time. The horizontal axis refers to the number of machines and the vertical axis to the computation time given in seconds. What can be observed here is the fact that there is not a speedup in each SOM size. In fact, the computation time of the smaller two (5x5 and 10x10) is increasing with an increase in the number of machines used for computation. This is mainly caused by the amount of communication, which increases with the number of machines. These two map sizes are simply too small to get a decrease in computing time via distributing its workload on several machines. The decrease in time of the 20x20 and especially the 40x40 map are on the other hand remarkable. This result confirms the claim, that the SOM algorithm can be distributed onto different machines in a network cluster in an efficient way.

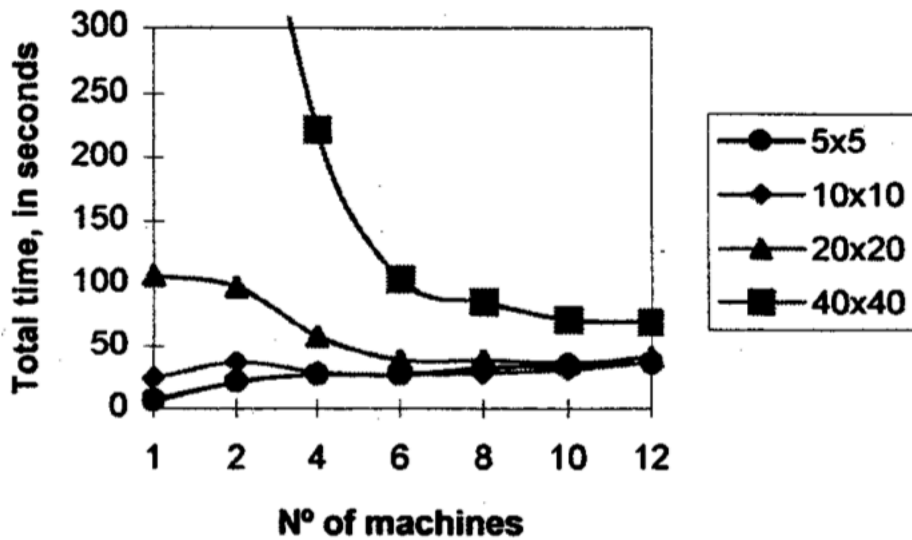

Figure 16 Curves of computation time (Bandeira, et al., 1998)

Table 1 also shows the execution times for each map size and each number of PCs used for computing as precise numbers. This can basically be seen as a numerical representation of Figure 16. The values within the table are again seconds. The fact, that the computation time can only be decreased down to a certain threshold through a rise of the amount of nodes within a cluster is shown here in more detail with the exact numbers. Through comparing the computation times with ten and twelve PCs one can see that this increase only lead in the biggest SOM to a further decrease in computation time. It will also be of importance to find the optimal number of cluster nodes for specific SOM and input data sizes for SOMatic.

	Size of map			
Nº of PCs	5x5	10x10	20x20	40x40
1	6	24	106	15268
2	21	37	97	623
4	27	29	58	222
6	27	27	39	103
8	32	28	39	85
10	34	31	36	71
12	36	37	41	69

Table 1 Execution times for different map sizes and numbers of PCs (Bandeira, et al., 1998)

3.2.2 A GPU accelerated distributed SOM implementation

Another approach to achieve a speedup is to run the computations on GPUs and distribute them (Wittek & Darányi, 2012). Within this research project a MapReduce-based (Dean & Ghemawat, 2004) implementation of SOMs is distributed on GPUs within a network size of eight nodes. These SOMs build on another project's (Sul & Tovchigrechko, 2011) batch formula of updating the weights. These weight updates are performed for each node and moved to the GPU with a matrix-based Euclidian distance matrix, which is faster than just computing this distance for each node. An algorithm (Li, et al., 2010) (van de Sande, et al., 2011) is used to decompose the steps into operations on the matrix level. Usually the computation of the Euclidian distance includes the square root. This operation is omitted here as it is time expensive to compute it on the GPU. As network cluster eight nodes from the Amazon Web Services (AWS) were used. AWS provides nodes, which are close in physical space and connected with a high-speed connection. The hardware of one node consists amongst others of two Intel Xeon X5570 quad-core central processing units (CPUs) with 23 gigabyte (GB) of memory and NVidia Tesla M2050, where one graphic card has 448 CUDA cores and 3 GB of device memory.

As software to distribute the algorithm on the nodes in the cluster the MPI (Snir, et al., 1996) and map-reduced message passing interface (MR-MPI) are used. To run the computations on the GPUs the NVidia CUDA (NVidia Corporation, 2014) framework was used. As input data about 35 million terms of a collection of about 84,000 PhD theses were analyzed. The number of dimensions was reduced to 200 via applying a random projection. As a result of the huge data size, it was only possible to run this implementation on all eight nodes, because the memory of the GPUs would otherwise not be sufficient enough. The size of the SOM had also been limited to 100 neurons. The execution time results and speedup factors are showed in Figure 17.

Method	Execution time	Speedup over CPU
CPU (64 cores)	2042s	-
GPU (16 Tesla)	433s	4.71x
CPU (64 cores) (One epoch)	1882s	-
GPU (16 Tesla) (One epoch)	194s	9.68x

Figure 17 Execution time comparison between GPU and CPU (Wittek & Darányi, 2012)

The GPU approach of distributing the SOM algorithm shows, that a remarkable speedup can be achieved when moving from CPU to GPU. However, it has a tighter limitation considering data sizes and it is easier to find and setup a network that uses CPUs for computation than GPUs.

3.2.3 Somoclu: A parallel library for SOMs

The name Somoclu is derived from "SOM on cluster" and is a parallel implementation of the SOM algorithm (Wittek, et al., 2016). It is written in C++ and can be used via a command-line interface or wrappers written in the programming languages R, Python and Matlab. With the command-line interface it is also possible to distribute the computations within a network cluster onto different machines. Somoclu builds up on two former implementations using MapReduce (Sul & Tovchigrechko, 2011) and distributed GPU computations (Wittek & Darányi, 2012). The MapReduce calls were replaced by MPI but within the multi-core process, MPI was replaced by OpenMP (Dagum & Menon, 1998), which increases its speed. Via OpenMP, each thread can work on the same SOM in contrary to MPI, where each thread holds a copy of the whole SOM. To implement the GPU kernel a library named Thrust (Bell & Hoberock, 2011) was used. The GPU implementation is not only using GPUs. The weight updates can be done in a faster way with CPUs because they can run an MPI process on each core whereas one GPU can only run one MPI process. The difference is that two GPUs have four times more data to process than eight CPU cores within the MPI processes. For that reason, the kernel is hybridized meaning the CPU cores are also used for the weight updates in the GPU implementation.

The distribution of the computations onto different machines within a cluster follows the master-slave architecture. A simple setup of this topology is displayed in Figure 18. The data is distributed to all nodes in an equal size. The finding of the BMU takes place without intermediate network communication at all as each search of the minimum distance is an independent process. The updates of the weights within the training stage require a two-way communication between the master and the slaves. Once the slaves are finished with their updates, they send their local changes to the master node, which accumulates them to the SOM. This new SOM is then distributed to each slave to compute the next training epoch. The change to a former distributed GPU computation (Wittek & Darányi, 2012) lies here in the use of ordinary MPI without the help of the MapReduce library.

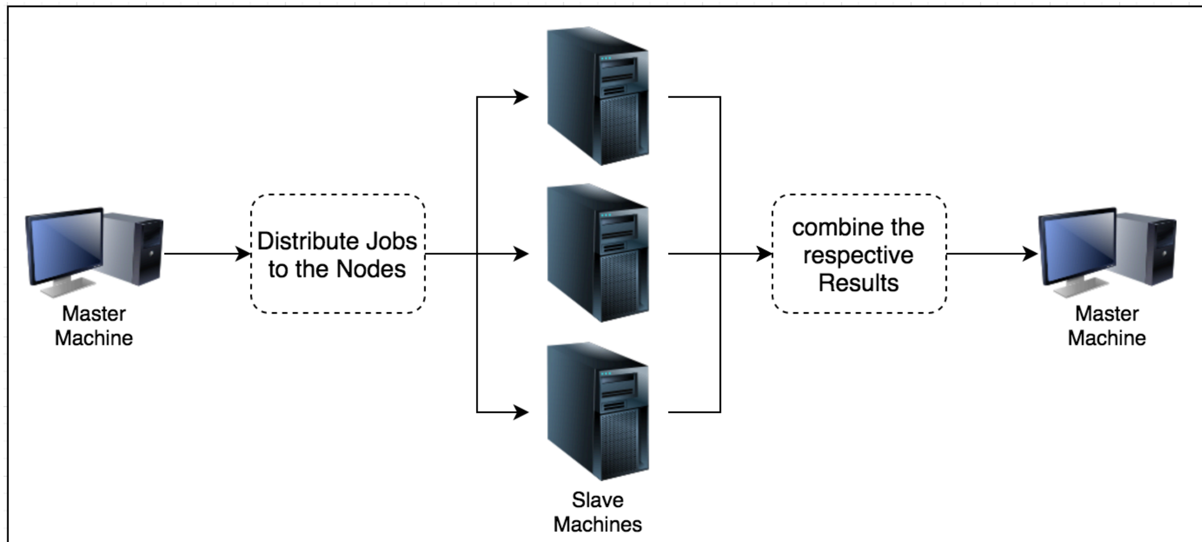

Figure 18 Overview of a master-slave architecture

To test the implementation a SOM with a size of 50x50 and 200x200 neurons was created. The number of the input vectors ranged from 12,500 to 100,000 and the dimensionality was set on 1,000. The random data instances consist of five percent non-zero values. Each machine in the network cluster had 22GB of memory, two Intel Xeon X5570 quad-core CPUs and two NVIDIA Tesla M2050 GPUs. The operation system was Ubuntu 12.04. The GPU implementation was about two times faster than the CPU version. This difference is not as high as only one GPU could be used as the Thrust template library is not efficient for two-dimensional data structures (Wittek, et al., 2016). The following Figure 19 shows the comparison between the CPU, the GPU and the single-core R package created by Kohonen (Wehrens & Buydens, 2007) on the SOM with 2500 neurons. Figure 20 shows the CPU and GPU time for the SOM with 40000 neurons.

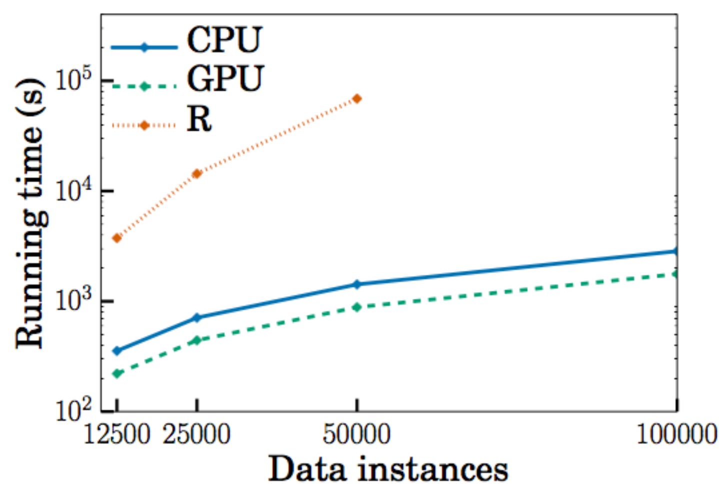

Figure 19 Runtime comparison on a 50x50 sized SOM (Wittek, et al., 2016)

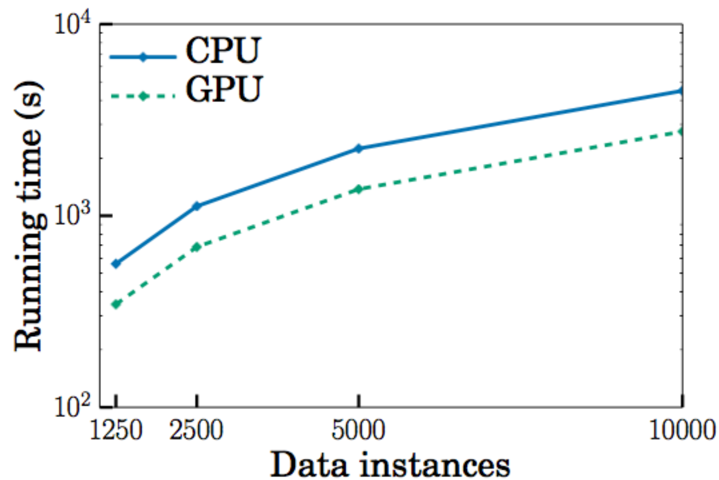

Figure 20 Runtime comparison on a 200x200 sized SOM (Wittek, et al., 2016)

The problem with updating the same neuron by different threads is said to be countered via a threshold presumably meaning that they cannot overlap each other. This should lead to a performance increase without a quality decrease (Wittek, et al., 2016). It is not mentioned how a possible update of the same neuron is handled, which can be solved for example with a lock of this neuron like implemented in the SOMatic trainer (Spöcklberger, 2013)

3.3 SOMatic

SOMatic can be divided into two parts: the trainer and the viewer. The trainer is a Java library that implements a SOM algorithm and applies a parallelized training phase (Spöcklberger, 2013). The viewer is used to visualize the created SOMs of the trainer with Processing (Rainer, 2013). Both were developed together in 2013 by a joint research project between SDSU in San Diego and CUAS in Villach by two students from Carinthia. The SOMatic trainer has a parallel training implementation but it is listed here in its own section together with the SOMatic viewer as SOMatic can be seen as the basis of this research project.

3.3.1 SOMatic Trainer

An important purpose of the development of the SOMatic trainer was to fit between self-contained SOM software and generic or outdated libraries (Spöcklberger, 2013). Therefore, it is implemented as a Java library and includes parallelized computations to be able to deal with bigger input data in a faster way. The training within this SOM application can be run on multiple cores on one machine to speed up the computation.

True parallelization of the computations can be achieved within a CPU containing two or more processing cores. The user can choose how many cores he wants to use when running the SOMatic trainer. Figure 21 shows the parameters, which need to be set to run the trainer (applied in the IDE Eclipse). The first two are strings and include the path to the input and output file. Parameters three and four define the size of the SOM, so for example if both of these values are 20, the SOM has a size of 400 neurons. The next parameter is the alpha value or also called learning rate factor. It controls the magnitude of the correction of the weights of the neighboring neurons (Lawrence, et al., 1999) and has to be between zero and one. The radius defines the influence size of the changes that are applied after finding a BMU. It usually decreases after each training epoch. The "iterations" value stands for the number of training runs and is in contrary to the radius usually increasing after an epoch. The user can also choose between the Euclidian, Manhattan, or Cosine distance, which is set with the integer "simMeasure". A one means Euclidian, a two Cosine and a three Manhattan. The last parameter defines the number of threads, which are used to do the training of the SOM.

```
static void runSOM(String sominputFilePath, String somoutputFilePath,
        int neuronNumberX, int neuronNumberY,float alphaValue, int radius,
        int iterations, int simMeasure, int nThreads) {
```

Figure 21 Parameters needed to run the SOMatic trainer

As one goal is to generate an additional output of SOMatic trainer in the form of a GeoJSON file, it is important to take a closer look at the in- and output data that the current implementation requires and produces. To be able to run this software, the input data has to adhere to a particular format. The file formats can either be comma separated value (CSV) or a SOM_PAK (Kohonen, et al., 1996) data file. In csv files, each line has to represent one input data item. Only the first row is not interpreted that way as it should contain the attribute names. The columns can be characters or numbers and include the attributes of each item. SOMatic can only deal with numerical numbers in the computation though but characters would not through an error.

The difference to the SOM_PAK data file is that the first line includes just one number, which represents the number of attributes. The attributes themselves are listed in the second line with "#atts" starting this line. The attribute names cannot be divided with spaces, so one needs to use underscores or a hyphen instead. As output, the SOMatic trainer produces two files: a codebook (.cod) and a .sprj file. The first one holds the information about the final SOM, which is similar to the data file and is also derived from SOM_PAK. The second one serves as information about the applied settings and parameters, which were used in this run. The following Figure 22 shows an example of the .sprj file.

```
### FILE PATHS ###
# the file that holdes normalized values that were used to train the SOM and can be mapped onto the SOM
trainingvectorfile = C:\Users\5Y5T3M4DM1N\Dropbox\SOM\Data\census_carinthia_2001\gem_k.dat
# the file that holds the original data
originaldatafile = C:\Users\fgkowatsch\Desktop\FGK\data\carinthiaCensus.dat
# the SOM file
codebookfile = C:\Users\fgkowatsch\Desktop\FGK\data\output_data\SOM_Trained.cod
### PREPROCESSING PARAMETERS ###
# defines how many attributes occur, each attribute will be described ina follow up line in the same order
numberofattributes = 43
#schema: normalizationmethod min max mean stdDev booleThreshold linearNormalizationOffset weight+
attribute0 = unnormalized 0.0 1.0 0.22608519 0.18558367 0.5 0.0 1.0
attribute1 = unnormalized 0.0 1.0 0.26959985 0.1559373 0.5 0.0 1.0
attribute2 = unnormalized 0.0 1.0 0.034244373 0.103525504 0.5 0.0 1.0
attribute3 = unnormalized 0.0 1.0 0.039963476 0.10542804 0.5 0.0 1.0
attribute4 = unnormalized 0.0 1.0 0.04189724 0.10651173 0.5 0.0 1.0
attribute5 = unnormalized 0.0 1.0 0.038329482 0.10453731 0.5 0.0 1.0
```

Figure 22 Example of a .sprj output file

The parallelization is a major interest of this research project and that is why it was also deeply analyzed in the SOMatic trainer (Spöcklberger, 2013) implementation. In general, it has two purposes: to speed up the training through the integration of more threads and to monitor its progress. An additional training surveillance thread (TST) is created and used to follow the advance of the training process within a progress bar object. Its main usage is to give the time progress as feedback to the user who can then better estimate when SOMatic will be finished. The following Figure 23 visualizes the sequence and purposes of the threads used within the training of the SOM.

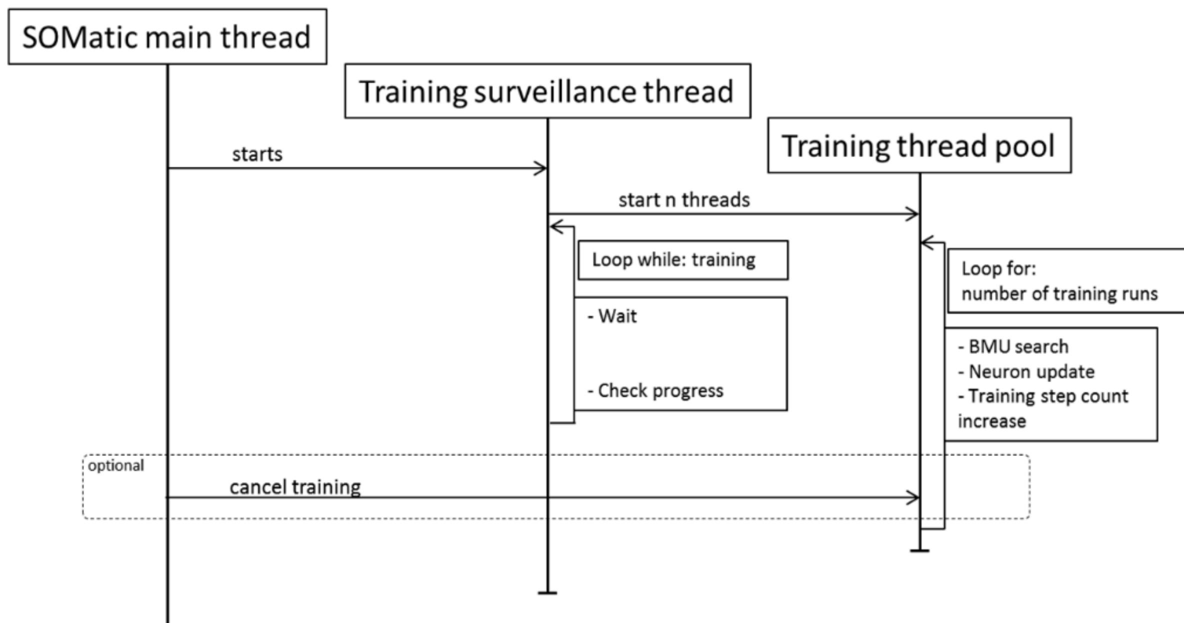

Figure 23 Sequence of the training threads within the SOMatic trainer (Spöcklberger, 2013)

Depending on the initial settings given by the user, a certain number of threads is created from a thread pool. The number of training iterations is then divided by the number of threads so that each thread has the same amount of work to do. Figure 24 displays the overview of how the parallelization is implemented in SOMatic. It shows how the computation within one step of the training process running 20,000 iterations is divided onto two threads. How one iteration looks like has been explained in Figure 3.

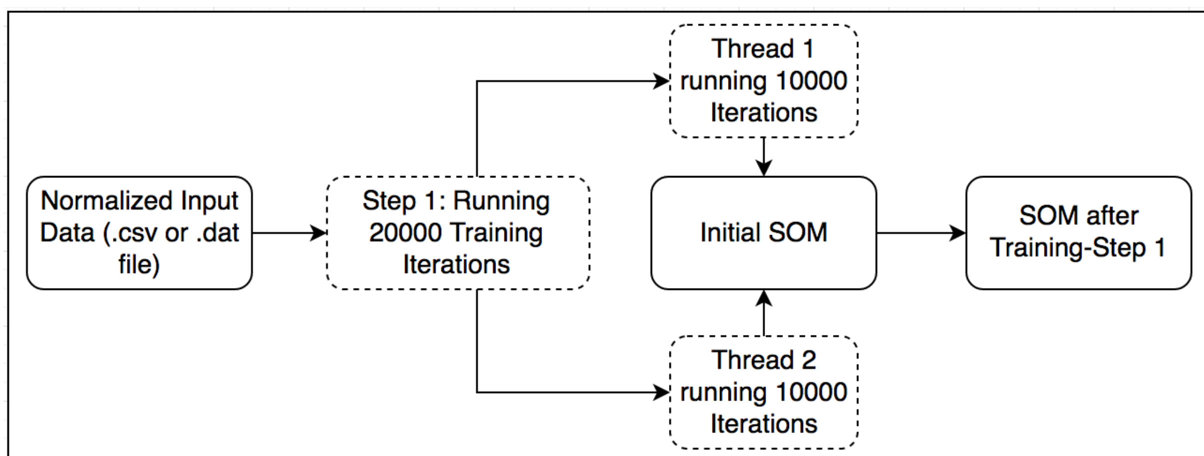

Figure 24 One training step parallelized onto two threads in SOMatic

The process of updating a neuron when two or more threads also seek to update the same neuron is still executed sequentially. This produces latency as the other thread(s) are forced to wait until the first thread finished the updating. Reading works in parallel, but the updating is a process of writing and therefore cannot be performed in parallel as this would lead to inconsistencies. Figure 25 gives a visual expression of the overlapping of the neighborhoods, where the latency issue could occur. The light-red and light-blue neurons represent the neighborhoods around two neurons (red and blue), which were found as BMUs by two processors. Neurons, which are situated in the shared neighborhood colored in purple can cause a delay of the weight-updating if both threads want to update the same neuron at the same time. This issue is more likely to happen in the early stages of the training when the neighborhood radius is still relatively high and/or when the number of threads that work in parallel on the SOM is higher.

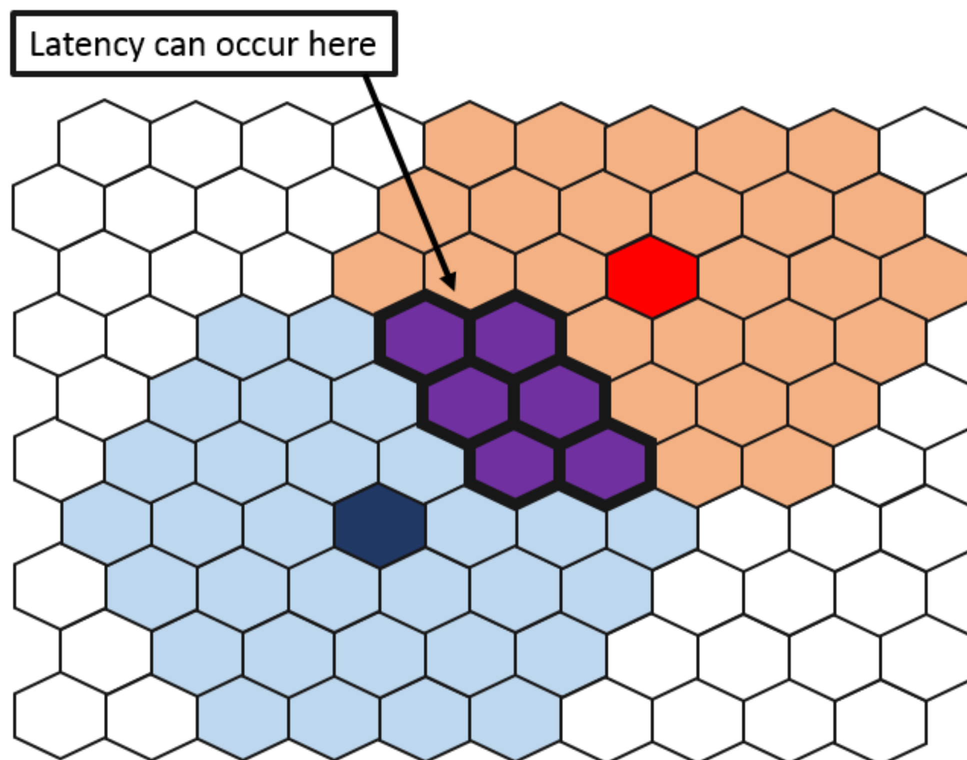

Figure 25 Neighborhood overlapping (purple neurons) of two BMUs (red and dark blue) in parallel training

The achieved speedup through using this architecture is almost proportional to the number of training threads. What delays the computation are the initialization and start of the threads from the thread pool, other processes running on that machine at that moment and the previously described issue with the simultaneous neuron update. The testing showed that through the parallelization it can be up to 9.14 times faster compared to a sequential training with only one thread.

To measure the quality of the output SOMs, a function computing the AQE is included within the SOMatic Trainer. It computes the difference in attribute space between each input vector and its corresponding BMU and averages the results. It was used by Spöcklberger (2013) and will be used in this project as well to be able to validate the quality of the resulting SOMs. Spöcklberger also created a graphical user interface (GUI) in Java, which is also usable in Processing so the user can run the trainer easier as a Java application, but also to show the compatibility of the library with other IDEs.

3.3.2 SOMatic Viewer

The SOMatic Viewer (Rainer, 2013) is an interactive SOM visualization tool which is implemented for Processing and also as standalone Java application. The user can choose between seven SOM visualization techniques like k-means clustering or component planes. To compute the SOMs, the viewer uses the SOMatic trainer (Spöcklberger, 2013). Before this project, there was no other SOM visualization software using Processing. The SOMatic viewer was developed along with the trainer and has therefore to be mentioned additionally when one is talking about SOMatic. This research is not concerned with the viewer because the focus lies solely in the computational aspects of the SOMs, which is implemented within the trainer.

4. Method of Solution

This chapter explains the overall project workflow in detail with charts to get a graphical view of the processes that will be implemented and how they interact with each other. It also shows the approach on how to solve the core problems and gives explanations on how it is planned to answer the research questions.

4.1 Procedural Workflow

The following Figure 26 gives an overview of the procedural workflow, which serves as the guideline how this project is organized.

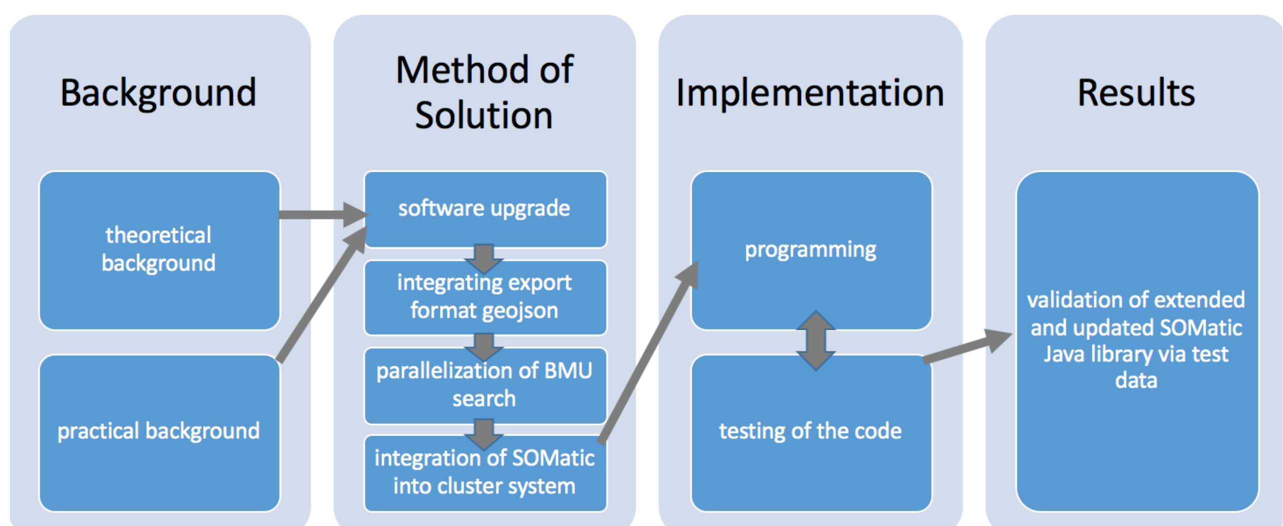

Figure 26 Project workflow overview

4.1.1 Background

The first step is to analyze previous case studies. This includes analyzing best-practice examples as well as taking a deeper look into the theoretical background, which is needed to conduct this research project. This step is required in order to answer the following questions: What implementations exist that use similar concepts and techniques? What approaches to solve the problems in this project could be found in these best-practice examples? What are the important technologies and techniques needed to conduct a research project in this field?

4.1.2 Method of Solution

This phase gives answers to the "how" questions in this project: How does the workflow of this project look? How will the code be updated and migrated to newer software versions? How can the original SOMatic output be exported in the GeoJSON format? How can one implement the parallelization of the BMU search? How will the integration of SOMatic into a cluster computing system look?

4.1.3 Implementation

This section is focused on implementing the planned steps in the method of solution. It explains the applied programming models and code, mostly in the Java language. To get a better understanding of the connections and preconditions within the software, flowcharts are utilized. The newly added methods are repeatedly tested with samples of data to ensure the persistence of its functionality. This answers the question: What are the programmed methods that are required to achieve the expected functionalities?

4.1.4 Results

The last step is to validate the programmed software and explain its functionalities via using data and visualizing it. This will be achieved with running the SOMatic Trainer 2.0 using multispectral image data and census data and looking at the time and error measurements. Therefore, it should answer following questions: To what extent did the usage and performance of SOMatic increase? What is the quality of the output?

4.2 Software Upgrade

4.2.1 From Java 6 to Java 8

The original SOMatic Trainer (Spöcklberger, 2013) was developed with Java 6. It used the system libraries as well as the Java runtime environment (JRE) of version six. The upgrading process is consisting of a change of all the system libraries as well as the used JRE to version eight. This can be done within the Eclipse IDE, where the library settings of the SOMatic Trainer Project can be modified.

After rebuilding the project, the IDE would highlight code that might have to be changed in order to be executable with the JRE eight. It is not planned to work through the whole code in detail and replace elderly (also called deprecated) code with newer classes or methods. The purpose of this migration is to assure that the code still runs using the latest libraries and runtime environment.

4.2.2 From Processing 2 to Processing 3

As the SOMatic Trainer library (Spöcklberger, 2013) is also used with Processing, it is critical to guarantee its compatibility with the latest Processing version as well. The upgraded Java library should still be executable within the Processing 3 IDE and code, which represent the latest version at this point in time. After applying the Java migration, the Processing migration will be carried out and tested as well.

4.2.3 Code Versioning and Cleanup

The starting point for coding within this project won't be the latest code version of SOMatic but the last one with the needed functionalities. Therefore, it will be necessary to walk through earlier versions of the code via the versioning tool Apache Subversion (SVN), which was used in the last implementation and will also be used in this one. Furthermore, each line of the code will be reviewed and those not needed or any unnecessary fragments will be deleted to reduce the size of it. Some comments may also be added to make it easier to understand for this implementation and possible future projects as well.

4.3 GeoJSON

One key aspect of the software enhancements of SOMatic is to include GeoJSON as a secondary output format apart from codebook files. It will be possible to visualize the trained SOM as well as the BMUs of this computation without any further processing in a browser or Geoinformation program like Quantum GIS (QGIS). Additionally, the trained SOM will be reusable as the GeoJSON output will be able to serve as input for another SOMatic computation. The reusability of the SOMatic output was thus far a bottleneck and will be improved through including a functionality to read-in the output in the GeoJSON format, additionally to the already existing function to read-in a codebook file. These functions will be programmed in additional methods in the Java code within the Eclipse IDE.

4.4 Parallelization

4.4.1 Local Parallelization: The BMU Search

The BMU search is a process in the SOM algorithm that is embarrassingly parallel as already described in section 2.2 in this thesis. This means, that all available computing resources can be used for computation and no communication has to take place between them during the process of computing the distance. As the SOMatic Trainer (Spöcklberger, 2013) uses a parallelization of the training steps and no batch computation, the parallel implementation described by Lawrence et al (1999) cannot be applied here.

A more effective way of searching the BMU can be achieved with a data-partition approach, like it is used in the batch computation of the SOM algorithm described in section 2.2.3. It is not applicable during training because of the different parallelization technique in the SOMatic trainer, but it can be used after the whole training process is finished. As input serve the final SOM which gets copied, and the input data vectors which get split up equally (or +/- some vectors depending on remainder) among the available, or by the user predefined, number of threads as shown in Figure 27 for the computation with two threads. The SOM gets copied onto both threads into their respective memory. The BMUs are then stored in a GeoJSON file, which holds only the BMUs, additionally to the second GeoJSON file containing all neurons of the final SOM. So the output of this BMU projection of new input vectors on a trained SOM will only consist of one new GeoJSON file holding the BMUs for the newly projected data.

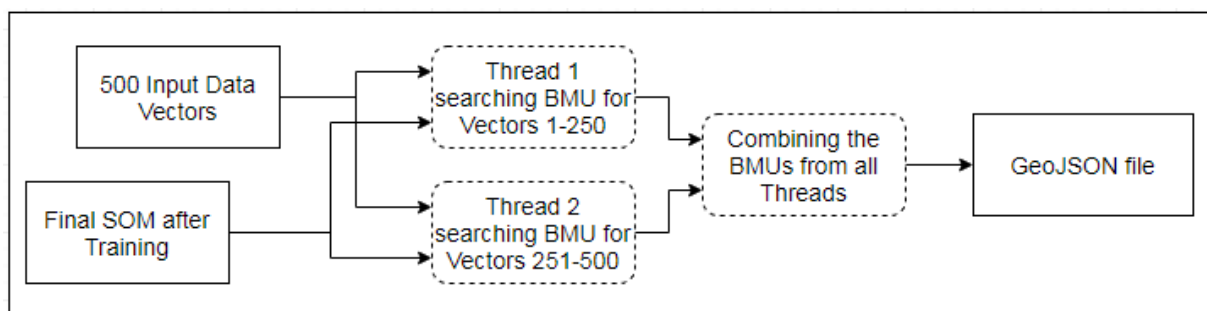

Figure 27 Example for Data parallelization approach in the BMU search

4.4.2 Distributed Parallelization: Cluster Computation in a Network

The following diagrams were created to be able to understand the principle components and steps that are behind the distributed computations using SOMatic on several machines in a network cluster. The system consists of two main components: the master machine, which controls and distributes the whole work and the computation nodes, also called "slave machines". The responsibility of each slave is to execute the job, which gets assigned to it. To be able to use this cluster, there are several prerequisites considering the software of the machines: To run the latest version of JPPF (5.2) it is required to have at least Java 7 as well as Apache Ant 1.7 installed and the environment variables in the settings set to the root folders of the respective installations. The "JAVA_HOME" variable has to point to the java development kit (JDK) installation folder (not to the JRE folder, otherwise it will not work) and the "ANT_HOME" variable has to point to the Ant installation folder. A copy of the SOMatic library is also needed at each machine. Furthermore, the network nodes should all have access to a shared file system to eliminate the time and amount that would be needed to transfer the data to and from each machine otherwise.

There are two different scenarios that will be implemented here: In the first one, all slaves run SOMatic on the same input data and SOM once and its resulting SOMs will then be combined after the computations are finished by the master machine. This will be used as an introducing step to JPPF. It is necessary to understand the basics behind using such a software to be able to continue with further developments. As shown in Figure 28, the steps of the first scenario are the following: The first one is to initialize a random SOM based on the input data. After that the slave nodes get that SOM as well as the input data and then each of them running the algorithm on copies of them, so all of them can do it at the same time. The last step is to combine the results.

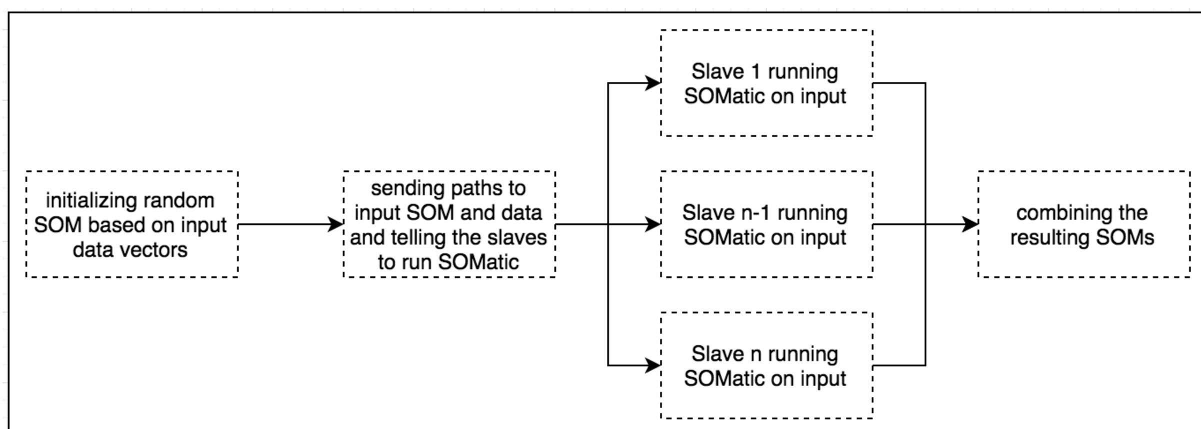

Figure 28 Scenario one of the distributed computation of SOMatic

The second scenario involves a random partition of the input data onto each slave and a repeated using of the SOMatic algorithm in a certain number of iterations. The combined SOM after one iteration serves as input for the next iteration. The following Figure 29 describes the second scenario in detail.

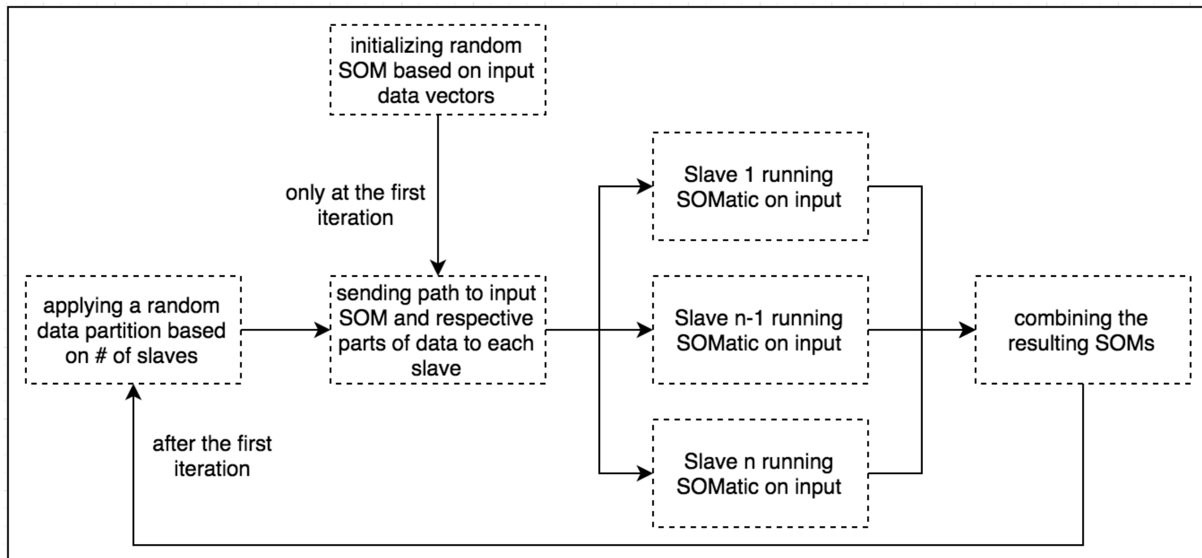

Figure 29 Scenario two of the distributed computation of SOMatic

The following workflow displayed in Figure 30 shows how the SOMatic Trainer could be integrated into JPPF. It has the same principles as the workflows shown in Figure 28 and Figure 29 but is more detailed by including the JPPF architecture with jobs and tasks. After executing the first four steps of the Trainer, namely reading in the input data, creating the training vectors, normalizing them and initializing of the random SOM, the next step is to create tasks and add them to a newly created job. The number of tasks should be the same as the number of nodes that are usable within the cluster. Until here, everything should be performed on the server.

When the job is executed via typing the command "ant" in the console at the location of the application, the tasks are sent to the nodes, which should then perform the training of the SOM. In the first scenario, they should only train it once by using the whole input data. After the tasks are finished, the SOMs are merged to one SOM, which is the final result here. In the second scenario, each task gets a different part of the input data assigned for the training. After the job is finished, the SOMs get merged and another job as well as new tasks are created, which have this time the merged SOM and a different distribution of the training data as input. The amount of iterations this process will take can be predefined by the user.

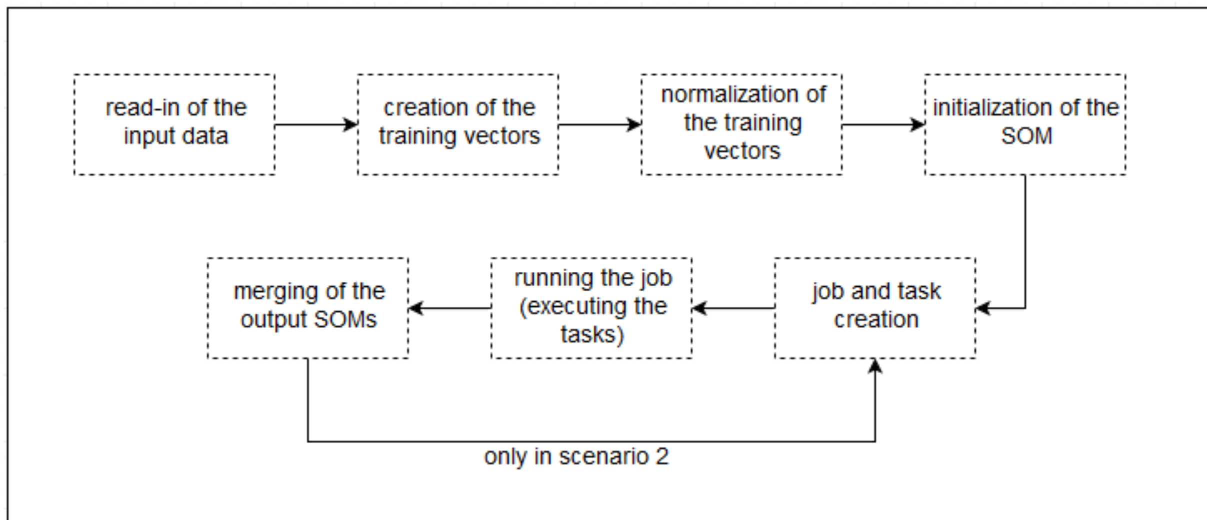

Figure 30 Workflow of the JPPF Implementation of the SOMatic Trainer

4.5 Software, Tools and Libraries

The main software that will be used within this project is the IDE Eclipse. It is an open source software that is widely used for programming, especially within the Java development world. Two versions of that software will be used in this project: Eclipse Neon in San Diego and Eclipse Oxygen in Villach. To keep a version tracking of the programming progress the open source software Apache Subversion (SVN) will be used. This is inevitable as the former implementation of the SOMatic Trainer (Spöcklberger, 2013) was also using this version tracking tool and therefore this project will continue with its usage. Another program that is also open source and is going to create the visualizations of the GeoJSON output is a known GIS called QGIS. The exact version of that software is 2.12 Lyon.

In the IDE Eclipse, there is one main tool that helps to control the dependencies from the Java code to different libraries. This tool is called Apache Maven and is directly implemented within Eclipse. Instead of a Java Project, the code will be written in a Maven Project. The dependencies have to be written in an extensible markup language (XML) file called project object model (POM).

It dynamically adds the libraries to the project without having the need to download and define the exact path to them in the build-path setting in the Project. Figure 31 shows the integration of the Processing library in the POM.xml file. The core library with the version 3.2.3 is referenced, as one prerequisite of the upgrade of the SOMatic Trainer is to be able to be compatible with Processing 3.

As a cluster computation framework JPPF will be used. The version 5.2 will be setup locally and on a machine cluster in San Diego, where the implementation will be tested on. To be able to execute a JPPF task, Apache Ant is needed as well, to be precise at least in the version 1.8 (which was used here).

```
24        <!-- https://mvnrepository.com/artifact/org.processing/core -->
25        <dependency>
26            <groupId>org.processing</groupId>
27            <artifactId>core</artifactId>
28            <version>3.2.3</version>
29        </dependency>
```

Figure 31 Example of a library dependency in a Maven POM.xml file

Apart from the Processing core library, there are only two other libraries (besides the Java system libraries), which are used for this Eclipse project: junit 3.8.1 and json-simple 1.1.1. The first one is automatically created in the POM.xml file and referenced by Maven. The second one will be needed to create the GeoJSON output.

When it comes to measuring the output SOMs of the SOMatic Trainer, the main tool to evaluate its accuracy is the AQE. It computes the difference between the attribute values from the input vectors to all their respective BMUs, adds them up and divides them by the number of input vectors to get the average. This gives also a good indication about how good a SOM describes the input dataset. The functionality of computing the AQE is already implemented within the SOMatic Trainer.

5. Implementation

This chapter describes the performed programming and how the methods are applied within this project.

5.1 Working Java Code

Before being able to start with programming the extensions one needs a working programming code. A version tracking tool called SVN was used when implementing the software and will also continually be in use. Via the command "checkout" different revisions where obtained and tested on their functionality in Eclipse. These revisions refer to different stages of the code.

The needed revision should have the same functionality as the jar file that was used so far to compute SOMs. To reach that goal, the newest version of the SOMatic trainer (where the Java classes were available) was compared with older versions. Through investigating the comments, which were written when a new revision was uploaded to SVN, it turned out that the needed revision should be around 70. After checking several versions, the code that matches the jar file was obtained at revision 74. This code served as the basis of this project.

To be able to handle the libraries in an efficient way, a Maven project was created where all the Java classes were imported. Furthermore, the folder structure of the classes, also called packages in Eclipse has been adapted so that it is more clear which classes are responsible for what part within the Maven project. The needed dependencies were added in the pom.xml file, namely the processing core v3.2.3 and later also the json-simple 1.1.1 libraries.

5.2 GeoJSON Output

As an addition to the codebook file created by the SOMatic trainer, the output is also stored in two GeoJSON files. To achieve that, the class FileWriter.java was extended with two methods: "writeSomToGeoJson()" and "writeBMUstoGeoJson()". The first one writes the whole SOM, so the neurons with their geometry plus the attributes after training into a newly created GeoJSON file. It creates the needed JSON objects and arrays and fills them with data retrieved from the geometry computation (details see section 5.3) and from the neurons, which hold the attribute values.

The second method is using the same geometry of the whole SOM but is only showing which neuron is the BMU for which input vector. This is done by indicating the corresponding input vector(s) in the BMU neurons. Each one of them gets a random point within the polygon assigned to it. This is needed in case two or more input vectors share the same BMU, so they would not be displayed one above the other, but get distinct coordinates assigned to them. Figure 32 shows how the polygon coordinates are extracted and written into JSON arrays. This piece of code is used in both GeoJSON writing methods. After getting filled with coordinates, the double array "cords" is walked over in a for loop, which starts at 0 and increments each step by 2 as two following values in that array refer to one coordinate pair of one hexagonal corner point. Each coordinate pair is stored in an own JSON array. These arrays are then stored in a JSON array called "innerCoords", which is then added to another array called "outerCoords" in the last line in the here represented figure. This structure is needed to be able to add for example the points of a hole within the polygon, which is defined within the GeoJSON compliances and therefore has to be compliant with them.

```java
// hexagonal coordinates from the current neuron
double[] coords;
// computes the hexagonal coordinates
coords = s.createHexaPolygon(currNeuron);
// walks through the coordinate pairs
for (int u = 0; u < coords.length; u += 2){
    // creates a coords pair and gets them from the double array
    JSONArray pointCoords = new JSONArray();
    pointCoords.add(coords[u]);
    pointCoords.add(coords[u+1]);
    // adds the coord pair to the coordinates json array
    innerCoords.add(pointCoords);
}
outerCoords.add(innerCoords);
```

Figure 32 Java code example showing the writing of the polygon coordinates into JSON Arrays

It is also possible to use a codebook file as an input and create a GeoJSON file from it. This is implemented in the method "fromCodToGeoJson()". It reads the parameters from the first, the attribute names from the second and the attribute values from the following lines in and creates a new SOM out of this data. Then this SOM is written to a GeoJSON file using the method "writeSomToGeoJson()".

5.3 Geometry of the Neurons

To compute an exact geometry for the neurons in a 2D coordinate system two functions had to be implemented. One function is to place each neuron at an x/y position in a coordinate system starting at bottom left and one to create hexagonal polygons around these coordinates. This is needed as it should be possible to visualize the content of the GeoJSON file immediately without any further processing. Basically each neuron is represented as a polygon with a hexagonal shape that has a midpoint and six corner points. The distance between two mid points of two neighboring neurons is one times the scaling factor, which can be defined by the user to enlarge or minimize the size of the visualized SOM.

The Cartesian point coordinates representing their centroids are computed via using the ID of the respective neuron and the size of the SOM on the x-axis. The x values are whole numbers starting at zero and going till the number of neurons on the x-axis. The y values are harder to compute as they need to be multiplied with a factor to fit in between the space of the two neurons below in every second row as it can be observed in Figure 33. This factor is one divided by the sinus of 60. In reference to the following figure, the centroid of the neuron with ID 0 has the coordinates **(**0**/**0**)**, neuron 1 has **(**1**/**0**)** and neuron 2 **(**2**/**0**)**. The neuron with the ID 3 in the second row from the bottom has the coordinates **(**0.5**/**1*1/sin(60)**)** and neuron 4 is at **(**1.5**/**1*1/sin(60)**)**. These exact coordinates are needed in order to clearly define the location of the corner points to be able to compute the polygonal representation of the SOM correctly.

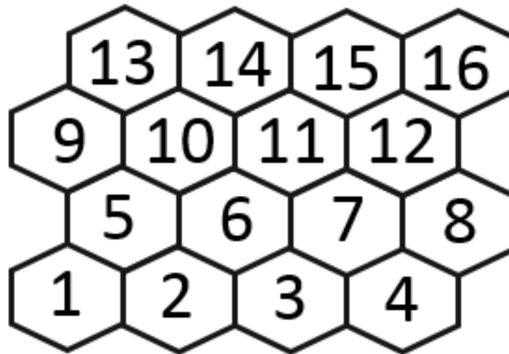

Figure 33 Hexagonal neuron alignment example with applied numeration

The polygon coordinates, respectively the corner points are basically created based on the centroid coordinates via using the rules of 30-60-90 triangles as illustrated in Figure 34. The angle of α is always 30 and the angle of β always 60 degrees. The first edge between α and β is always two times the second edge between the right angle and β (called x). The third edge between the right angle and α is always the second edge times the square root of three.

These characteristics of this triangle allow it to compute all edges by only knowing the length of one of the three without the need of using a sine, cosine or tangent function. That makes it easier to implement and helps to reduce the computation time. The connection is made via using the third edge, which is half the distance between two neighboring neurons, or 0.5 times the scaling factor.

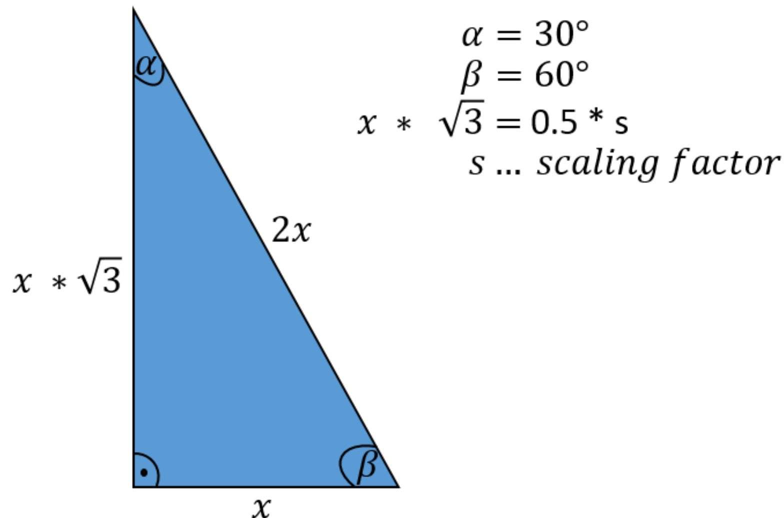

Figure 34 Characteristics of a 30-60-90 degree triangle

Figure 35 shows the 30-60-90 degree triangle used in a hexagon. The coordinates are computed using mainly two values: the distance from the centroid to an edge, which is 0.5 times the scaling factor (s), or x*√3 because the length between the centroids of two directly neighboring neurons is always 1*s. This distance is then divided by the square root of three to get the distance from a corner point to the middle of an edge (x). Then it is just an adding or subtracting of these values from the coordinates of the centroid to compute the coordinates of the hexagon starting at the top and moving counter-clockwise around. The first point has to be added again and the points have to be counter-clockwise in the array to match the specifications of the GeoJSON standard format. If the in Figure 35 shown neuron has the ID 0, the coordinates with a scaling factor of 2 for point A are **(**0**/**1.15**)**, for B **(**-1**/**0.58**)**, C **(**-1**/**-0.58**)**, D **(**0**/**-1.15**)**, E **(**1**/**-0.58**)** and F **(**1**/**0.58**)**.

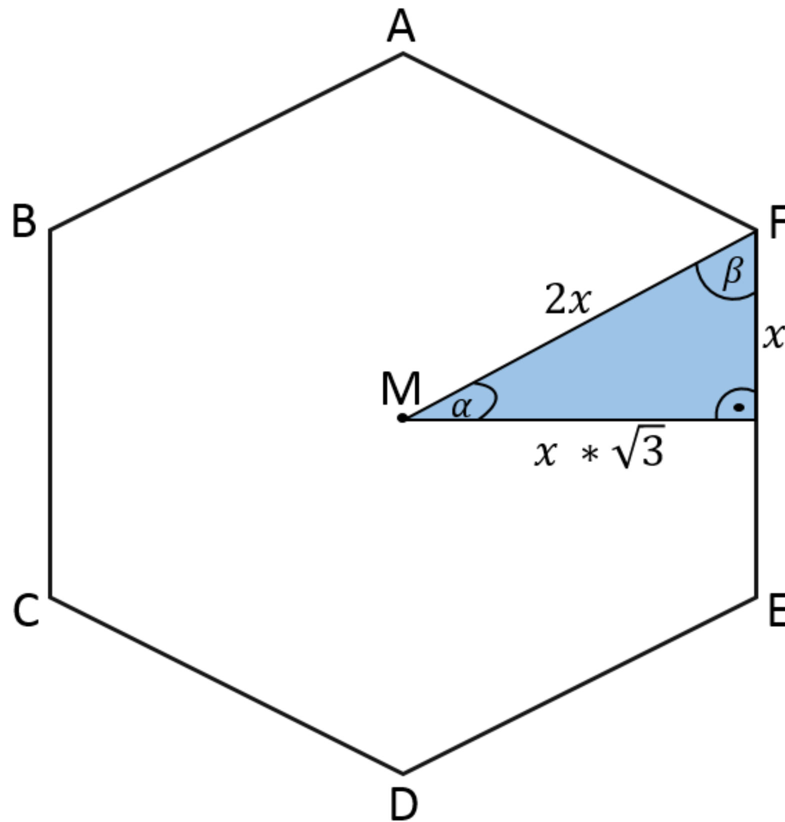

Figure 35 Usage of the 30-60-90 degree triangle characteristics within a hexagon

5.4 Integration of a parallelized BMU Search after Training

The functionality to apply a BMU search after training was not existent thus far. The here mentioned BMU search is not applied during training, but afterwards when other data, which was not part of the training data, needs to be projected onto a particular SOM. It can exploit the full parallel computation possibilities of the machine that is used in this scenario.

The user can define via a variable how many processors he or she would like to use to make a BMU search with the defined dataset onto the trained SOM. This option is still available as the setup and initializing of the different processes consume some time as well. If the dataset is not that big, it is not necessary to initialize many processors.

This parallelization follows the common data-partition method. The input data is read in, normalized and then split up onto a defined number of parts, which represents the amount of processors that will be used here. This is easy when there is a certain number of input vectors that can be divided by a certain number of threads without a remainder, for example having 20,000 input vectors and four threads.

When there are some remaining input vectors, the code gets a bit more complicated. Figure 36 shows the part of the code that handles the division of the input data and the issue with a possible remainder. “nTVs” is the number of input vectors, which gets divided by the number of threads “nThreads”. That quotient gets then saved in the variable “sParts” in line 206. As the data type of “sParts” is an integer, it is always a whole number without a remainder. The size of the remainder is computed via subtracting the number of input vectors with the product of the number of divisions times the number of threads in line 208. The parts of the input data get than copied into an array of training vectors in line 212 stepwise for each thread with the precomputed size and copied into an arraylist called “tvParts”, which stores all the parts of the input vectors. The second half of the code example is only used when there is a remainder, which is checked in line 217. In the lines 221 and 222 the last two arrays are removed as there was an additional one created in the for-loop with the size of the remainder and the second last one will be changed. This array will get the last full share plus the remaining training vectors and be added again to the arraylist. This can be observed in the lines 224 and 226.

```
204        // divide the tVs through number of available threads
205        // to get the size of the array parts
206        sParts = nTVs / nThreads;
207        // the remaining TVs (if any) will be added to the last thread
208        remainder = nTVs - (sParts * nThreads);
209
210        for (int i = 0; i < nTVs ; i += sParts){
211            // copies that part of the array to a new array and adds that to the arrayList
212            TrainingVector[] partialTVs = Arrays.copyOfRange(tVs, i, i + sParts);
213            tVParts.add(partialTVs);
214        }
215
216        // if there are remaining TVs
217        if (remainder != 0){
218            // remove the last 2 arrays from the arrayList because
219            // there is one additional array created with the size of the
220            // remainder in the last for loop + the last one without remainder
221            tVParts.remove(tVParts.size()-1);
222            tVParts.remove(tVParts.size()-1);
223            // adds the remaining TVs to the last partition
224            TrainingVector[] lastTVPart = Arrays.copyOfRange(tVs, nTVs - (sParts + remainder), nTVs);
225            // adds it again with the remaining TVs
226            tVParts.add(lastTVPart);
227        }
```

Figure 36 Code-excerpt of the parallelized BMU search after training

Every processor walks through its respective part and searches the BMU for each input vector. The code for this computation is the same as for the BMU search in the training process. The information which neuron is a BMU for which input vector is stored in the Neuron Java object and then readout when exported into the BMU GeoJSON file. The output of this computation can only be stored in the “bmus.geojson” file as it does not change any attributes but shows the BMUs for a given set of input vectors on an already trained SOM.

5.5 Cluster computing of SOMatic

Before being able to start programming, some prerequisites had to be fulfilled. The first step was to get a get a local setup of the needed software, namely JPPF and Apache Ant. The build paths for Java and Ant had to be set to the bin folders of the named software to be able to run JPPF via the command shell. Once everything was setup, the next step was to get the tutorial from JPPF (JPPF.org, 2017) up and running locally. This was relatively easy with the help of using the online documentation. The code of the tutorial can also serve as a starting point for individual development, which was the case here. The tutorial was copied into a newly created folder called somatic.

There were three libraries, which were needed here: processing-core.jar and json-simple.jar to be able to use the SOMatic Trainer and of course the Trainer itself, so somatic_2.0.jar. These three were copied into the lib folder of the JPPF somatic project folder. It also contains a source folder with two classes: Task.java and Runner.java. The task is created within the runner and sent to the nodes when a job is executed. The runner is running on the server and serves as the main control class that creates a job and task objects, executes them and processes the execution results, which get sent back by the nodes.

The code itself was setup as explained in Figure 30 in section 4.4.2. It could not be tested because the execution did not work at all in the first runs. After several error solving, it ran through but did not produce any output. The problems are documented and explained at the end of the next section 5.6.

5.6 Challenges/Problems

The first challenge in the implementation phase was to find out which revision from SVN was the wanted one to get a starting point for the programming. The most recent version was not suitable as it was not rounded up but had unfinished functions, which caused the SOMatic trainer to not run through properly. Therefore, the decision was made to use that revision as a starting point, which is reflected by the jar file used by Dr. Skupin in recent years.

The first step was to work through the notes provided at each revision to get an overview of the implemented functions. As an additional support, the available classes in the jar file were compared to those in prior SVN revisions. It turned out that the needed classes were available after revision 70. So this one was retrieved and tested in Eclipse. The output in the console was compared to the output of using the jar file.

They were not the same and so the next step was to compare the code itself from revision 70 to the newest revision via Tortoise Merge, a tool from SVN to be able to make such comparisons. Through a stepwise copying of newer code parts and testing them in Eclipse, the wanted functionality should be achievable. This method was not working though and therefore, it was thrown over again.

The next approach to get a working Java code was checking out different revisions and running them in Eclipse. Starting at revision 71 and going upwards, each one was checked out and tested in Eclipse. Revision 74 turned out to be the one that matches exactly the functionality and output of the working jar file. This was checked with comparing the processing outputs in the console in Eclipse, as well as the resulting codebook files.

The second bigger challenge was to get the SOMatic trainer output in a GeoJSON standardized format like it is explained by Butler (2016). The JSON has to be written and organized in a certain way to be able to be GeoJSON compliant. Apart from having the needed JSON objects included like "type" or "features", it also matters what is stored in a JSON object and what in a JSON array. The "features" contain one JSON array, but the "properties" of the objects within that array are only written in one JSON object and cannot be written in an Array. The following Figure 37 shows this specification with an example from the spec sheet (Butler, et al., 2016). One has to look at the "[" bracket after "features", which indicates the start of a JSON array, whereas the "{" bracket indicates a JSON object.

```
"type": "FeatureCollection",
"features": [{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [102.0, 0.5]
    },
    "properties": {
        "prop0": "value0"
    }
```

Figure 37 GeoJSON example (Butler, et al., 2016)

Another criteria was to write the coordinates of the corner-points of a polygon counter-clockwise into the object. This was met with initial issues, but was later resolved with some changes in the sequence of the corner-points computation within the Java code. Additionally, the first point has to be the same as the last point, so the polygon is closed again. This point has to be twice in the array and it has to have the same coordinates.

The work with a cluster computation, where the goal is to achieve a distributed computation using different machines turned out to be a bigger challenge than expected. What did not come out through the literature research and methodological considerations in earlier stages of the project was the need for having serializable objects in a distributed environment. Only then it is possible to pass Java objects, like a SOM object for example, between the server and the nodes. An attempt to fix that was made through adding the "implements serializable" at the header of each class but that alone was not enough. After writing that piece of code to each class it ran through without an error but the training did not work as there were no time differences when changing the parameters and the resulting SOMs from the nodes where empty as well. The method to control the parameters with using a singleton design pattern in the class "Global.java" is not applicable within a distributed environment, where more instances would be necessary.

Due to the prolongation of the implementation of the GeoJSON functionality, there was not much time left at the end for solving the issues that came up in the programming and testing of the cluster computing functionality. This does not mean that the problems were unsolvable but due to a lack of time and the need to round the project up and continuing with the thesis itself, it was not possible to finish this implementation step.

6. Results

Here, the outputs of the project as well as the applying of the SOMatic Trainer 2.0 onto two test datasets with the creation of a time and quality evaluation are shown and explained.

6.1 SOMatic Trainer 2.0

The following Figure 38 displays a class diagram showing the classes that are now included in the newly SOMatic Trainer 2.0 as well as their corresponding packages. Some classes were renamed and moved into other or newly created packages to better fit the overall structure of the Maven project in Eclipse. There are six packages containing a total of 16 classes. Newly added were only the two main classes "Main" and "TestMain". The other 14 were already existent in the original SOMatic Trainer. Furthermore, not all of the attributes and functions of the different classes are included here, only the most important ones. The following classes were modified or extended within this project: all classes in the packages "org.somatic.entities" and "org.somatic.main" and the classes "SOMatic", "Global", "FileReader" and "FileWriter". The other classes were not changed and therefore are the same as in version 1.0.

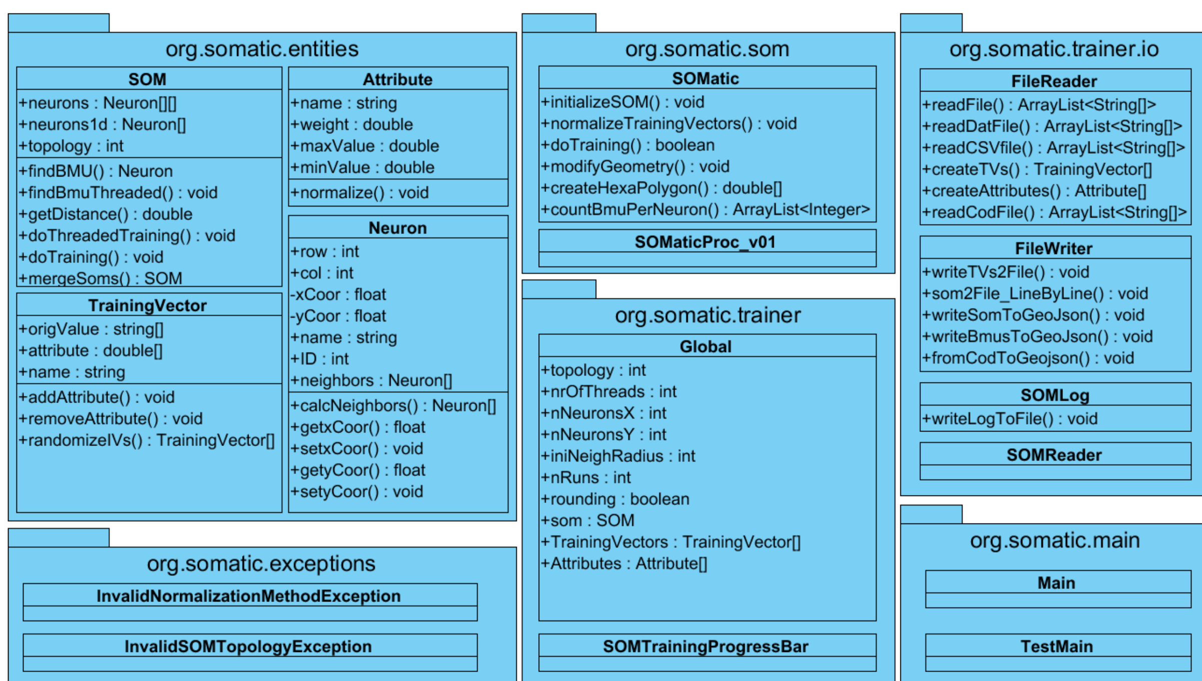

Figure 38 Class diagram of the SOMatic Trainer 2.0

The whole project is using the Java runtime engine (JRE) eight and its system libraries, meaning it is compatible with the latest Java version, as well as with the latest Processing version, which was also tested within the IDE having the same name. The class "FileWriter" is now able to export two GeoJSON files. The first GeoJSON file contains the neurons after the training is finished. Each neuron is represented by one GeoJSON object holding the attribute values as well as the geometry of the neuron as a polygon. How the geometry is computed is explained in the chapter 5.3. The attribute values are extracted from the trained SOM and written alongside its respective attribute name into a GeoJSON object. The second GeoJSON output file contains those neurons of the SOM that are the BMU for one or more of the input vectors. These neurons are of special interest as they show where the input data is located in the trained SOM. This GeoJSON file contains as many neuron objects as there are input vectors. The information which neuron is the BMU for what input vector is stored in the Neuron object in the code and extracted from there. Visualizations of using these GeoJSON output files are shown and explained in the following chapters 6.2.1 and 6.2.2.

The SOMatic Trainer 2.0 allows a projection of data onto an already existing SOM via a parallelized BMU search after training. This function is opening the door to visualize data that was not part of the training without redoing the whole training process. At the end of the chapter 6.2.3 the results of using this functionality on test data are shown and explained in detail. The distribution of the computation processes onto different machines within a network cluster could not be finished within the scope of this project. The progress of the implementation of that part and the faced challenges are explained at the end of chapter five and also discussed in chapter 7.

6.2 Applying SOMatic on Test Data

6.2.1 Carinthian Census Data

The first dataset that is used for visualizing the GeoJSON output is census data from the Austrian federal state Carinthia. It has 43 attributes representing various categories like size, or number of people in different age and sex categories. There are 132 input vectors representing all municipalities inside this dataset. It was also used to present the result of the first implementation of the SOMatic Trainer by Spöcklberger (2013). The following Figure 39 illustrates the BMU GeoJSON file, visualized in QGIS and showing the names of the municipalities. It was trained with the following training parameters: neurons on x = 30, neurons on y = 30, alpha value = 0.04, radius = 30, iterations in first run = 100000, similarity measure = 2, number of threads = 4, rounding = true, scaling factor = 2, training runs = 3, iterations in second run = 200000 and iterations in third run = 500000. The value 2 at the similarity measure means that the Cosine similarity was computed. The radius was decreased in the second run by 50% and in the third run by 80%. The machine on which these parameters were applied and that created the visualizations has 16 gigabyte of RAM and an Intel Xeon E5-1603 processor with four cores and threads and 2.8 GHz.

The in Figure 39 shown SOM represents a part of the content of the "bmus.geojson" output file, which is created by the SOMatic Trainer. What it is not showing is the ID and the vector count of each neuron. Every neuron knows for which and how many input vectors it is the BMU. Apart from adapting the colors and displaying the input vector names, nothing else was changed after the GeoJSON file was loaded into QGIS. There are two things that can be mentioned here from just looking at the result. The first one is the fact that cities like Klagenfurt, Villach and Wolfsberg are in the top-right region of the SOM. Therefore, they are different from the other municipalities, which can easily be explained by their bigger size in population. The second thing that can be pointed out here is the closeness of the municipalities Trebesing and Kleblach-Lind on the left outline of the displayed SOM. They are both smaller municipalities in two different valleys but have apparently a lot in common.

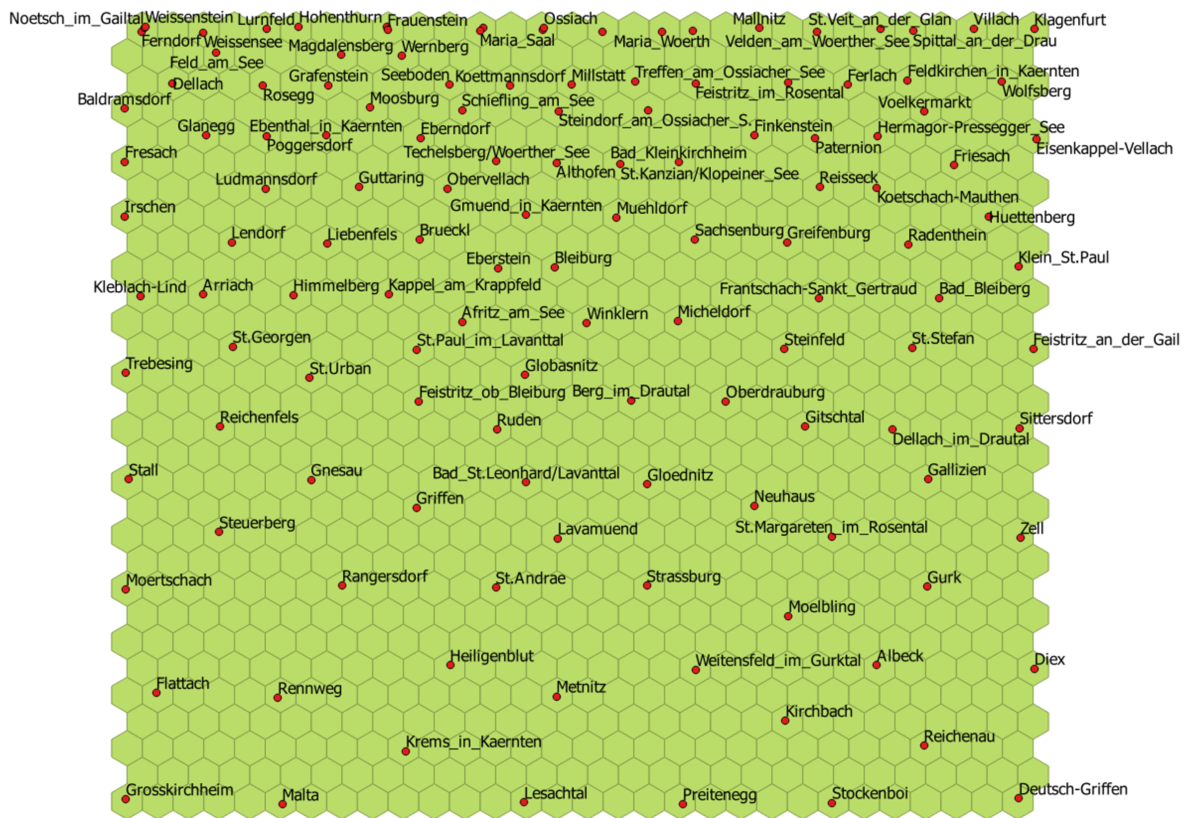

Figure 39 Carinthian census data: BMU GeoJSON visualization from QGIS

The next two figures will show visualizations from the distribution of two attribute values over the SOM. Figure 40 visualizes the distribution of an attribute giving the ratio of the Austrian citizens within each municipality. The classification method is natural breaks and it uses eight classes. One can identify a general decrease in the ratio of Austrian citizens when moving from bottom-left and left to top-right. This can be explained by the fact that cities, located in the top-right of the SOM, are usually more international than rural areas. Almost a complete inversion of the colors of the neurons can be observed when looking at the attribute that describes the ratio of non-EU citizens displayed in Figure 41. Again eight classes and the natural breaks classification method were used to create this visualization.

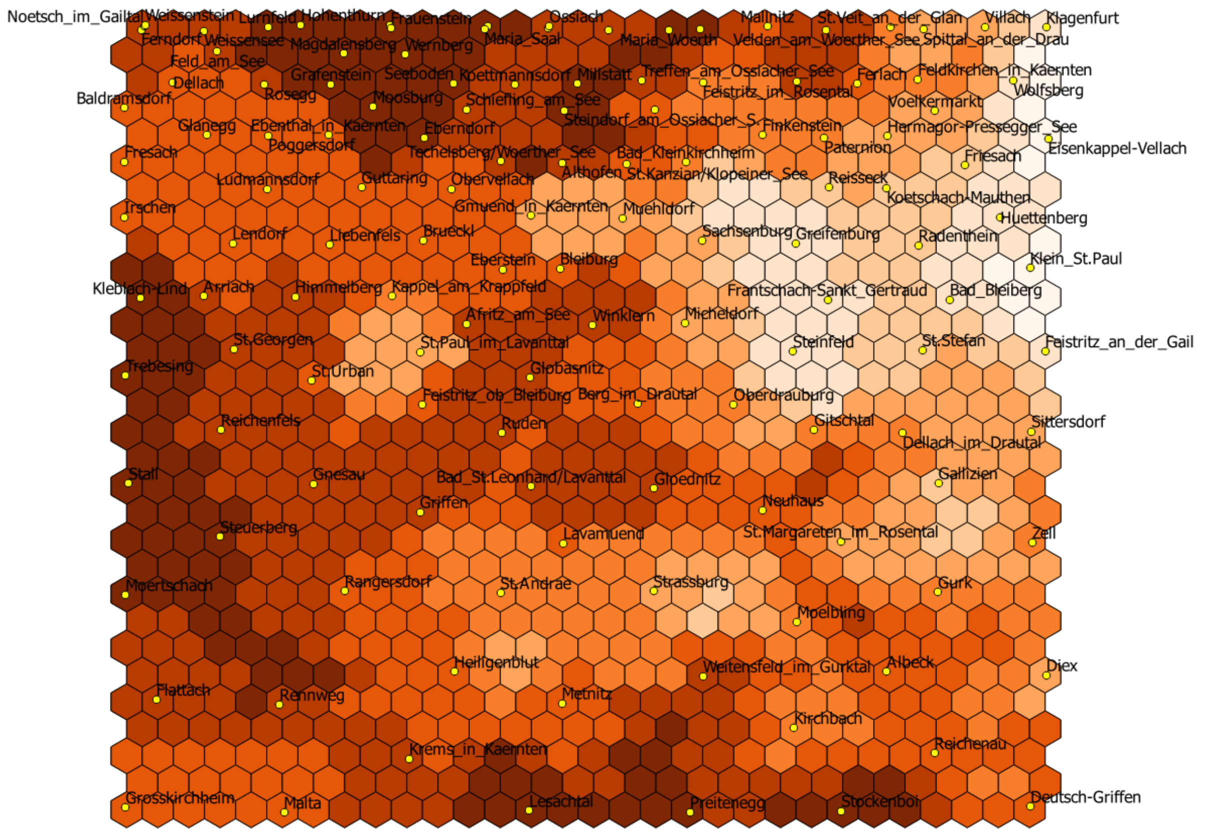

Figure 40 Carinthian census data: Distribution of ratio of Austrian citizens

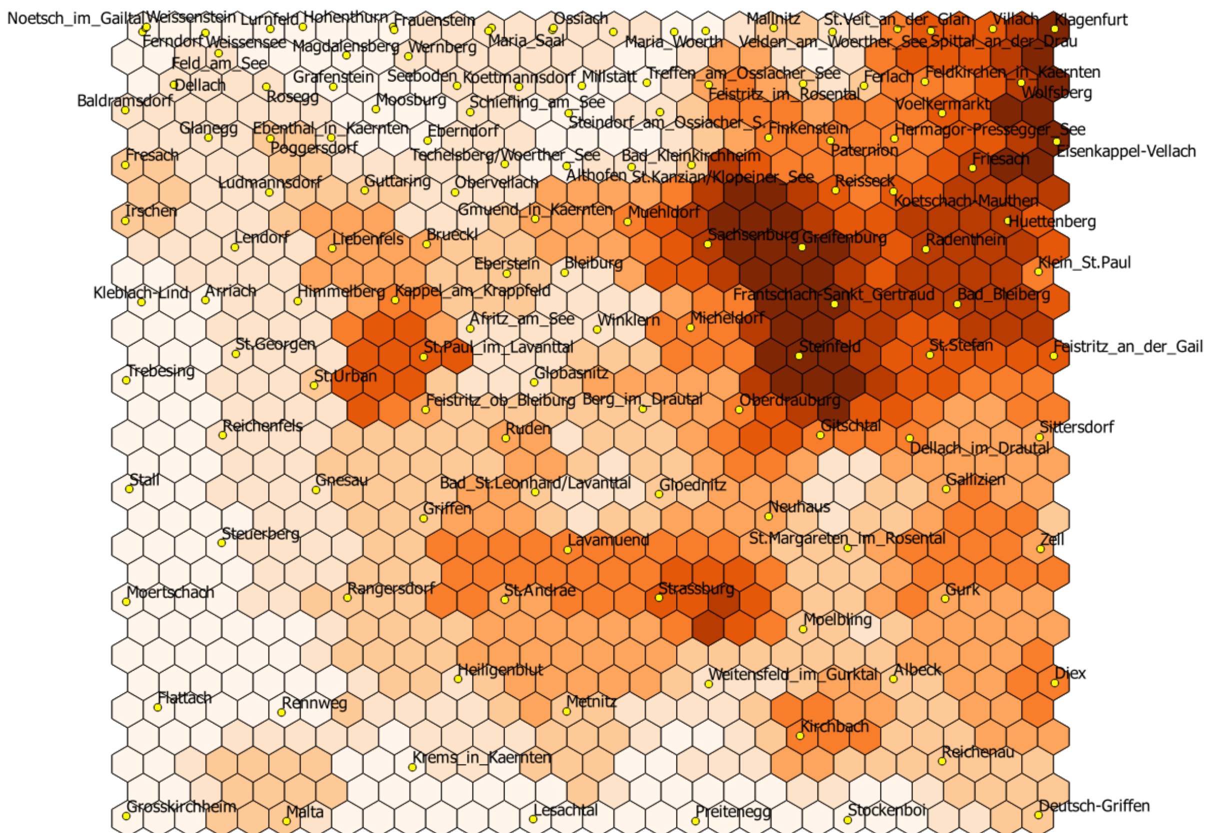

Figure 41 Carinthian census data: Distribution of ratio of non-EU citizens

6.2.2 Multispectral Image Data

The second dataset, which was used here in order to run the SOMatic Trainer and test its capabilities is multispectral image data. It is derived from the Landsat Thematic Mapper and consists of six color bands for three different years (1993, 2003 and 2013) always taken in April. The region, where the images were taken, is situated in the north of San Diego and called Batiquitos Lagoon. The following figure shows the three images in their true colors. Multispectral data can be used to detect and visualize changes in a better way. One can see on these three images going from left to right that the man-made objects like houses are increasing, especially in the top-right region of the images.

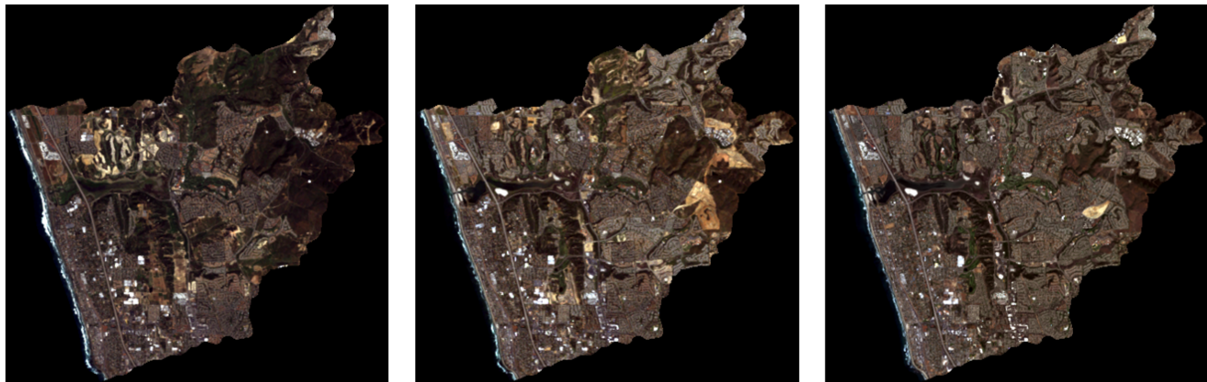

Figure 42 True color images from the Batiquitos Lagoon in the North of SD showing this region in April 1993 (left), 2003 (middle) and 2013 (right)

Looking at the multispectral data, the first band represents the blue spectral channel with a wavelength of 0.45 to 0.52 micrometers, the second one the green channel with a wavelength of 0.52 to 0.6 and the third one the red channel with a wavelength of 0.63 to 0.69, the fourth one the near infrared going from 0.76 to 0.9, the fifth one the shortwave infrared one from 1.55 to 1.75 and the sixth band containing the shortwave infrared two with a wavelength from 2.08 to 2.35. This makes 18 attributes for each of the 90955 pixels of which these images consist. Each band was normalized to a scale from zero to one, where the lowest value becomes a zero and the highest one a one.

This dataset was only used for testing the computation times and quality measurements of the SOMatic Trainer, which is described in the following chapter 6.2.3. It would be possible for example to create GeoJSON visualizations from it and use them together with the input pixels to make the changes within the different time slices visible, but this analyzation was out of the scope of this project.

6.2.3 Quality and Performance Evaluation

When it comes to the quality and correctness of a SOM, there are several methods that one could use to measure them. If some characteristics of the input data are already known in advance, then they can be used as a rough measure, as also Spöcklberger (2013) was mentioning in his thesis when he was using the same data source. Looking at the Carinthia census data again, the two biggest cities Klagenfurt and Villach are quite different from the other municipalities because they have about 60.000 and 100.000 inhabitants, which is more than six times of the mean of the other ones. So these two cities have to be somehow close to each other in the resulting SOM, which is the case in the given visualizations shown in 6.2.1. Another measure for the quality of a SOM is the AQE. The following

Table 2 shows the number of the run, the needed computation time in milliseconds and the AQE value for ten runs using the same parameters as described at the begin of chapter 6.2.1. The time includes the training of the SOM, the computation time for the AQE and the time it needs to write the three output files, so the two GeoJSON and the codebook file. The mean values show an average runtime of almost 54 seconds with an average AQE of 0.00137. The table and also the diagrams following later on have commas instead of points because of the German Microsoft Office software. The differences in the time and AQE values between the different test runs are quite low because the machine to run SOMatic was not running any other program than Eclipse.

nRun	Time	AQE
1	54973	0,001270532
2	53576	0,001464563
3	53596	0,001306308
4	53651	0,001645782
5	54148	0,001288336
6	53638	0,001289085
7	53591	0,001678092
8	53425	0,001388482
9	53984	0,001338515
10	53928	0,001060707
Mean	53851	0,00137304

Table 2 Mean of the computation time (in ms) and AQE for ten computation runs using the Carinthia census data

When the AQE values are compared to the results of Spöcklberger (2013) using the same data, there is a significant decrease form the former computation to the one presented here. It is not known which parameters he was using, so this might be the reason why they are different. Spöcklberger was also comparing the AQE values he got with SOMatic to the ones from SOM_PAK. The lowest value is at 0.11 when using SOMatic with four threads using 10 million training runs. The here shown values were computed by using 800,000 training runs in three training steps. This shows that the AQE is quite low meaning the quality of the output SOM is relatively high.

The next tests with the Carinthian census data were done with the same parameters, except for the number of threads, which was changed to one, two, four, six, eight and sixteen. The computation time and AQE was measured again. For using six and more threads, the machine had to be changed in order to be able to test it properly. A server in CUAS was used, which has two Intel Xeon E5-2620 v4 CPUs with 2.10 GHz and eight cores each allowing to run 32 threads in parallel. In theory, it should be possible to run 32 threads with this hardware but only up to 16 threads could be used in Eclipse. Each thread number was run ten times. The mean values out of these computations were used to create most diagrams in this section. For some visualizations, the median had to be used as the mean would have been influenced by some outliers, which were most likely caused by some background computations on the server.

Figure 43 visualizes the computation time in milliseconds with the corresponding speedup for six different numbers of threads. When looking at the blue line, one can see a clear decrease in the computation time with an increase in the number of threads. This decrease is higher at the beginning but slowing down with increasing numbers of threads. When using one thread the training took 280558 milliseconds, which is about four minutes and 41 seconds but with sixteen threads, the time was reduced to 26 seconds. The grey line is showing the time speedup in percent in reference to the usage of one thread. The biggest step is from one to two threads with almost 60 percent speedup. The highest speedup in the test runs could be achieved with sixteen threads, where it is at 91 percent. The curve is getting smoother with a higher number of threads and will presumably go down again as the time of setting up the threads and creating the partitions is also increasing. The drop in the speedup, respectively the increase in the computation time from four to six threads lies behind the change of the machine. This could by caused by a slower processor in the server compared to the local machine when using a lower number of threads. The connection to the server was established via a local machine but this should not have a negative effect on the computation time as this was measured in Eclipse running at the server.

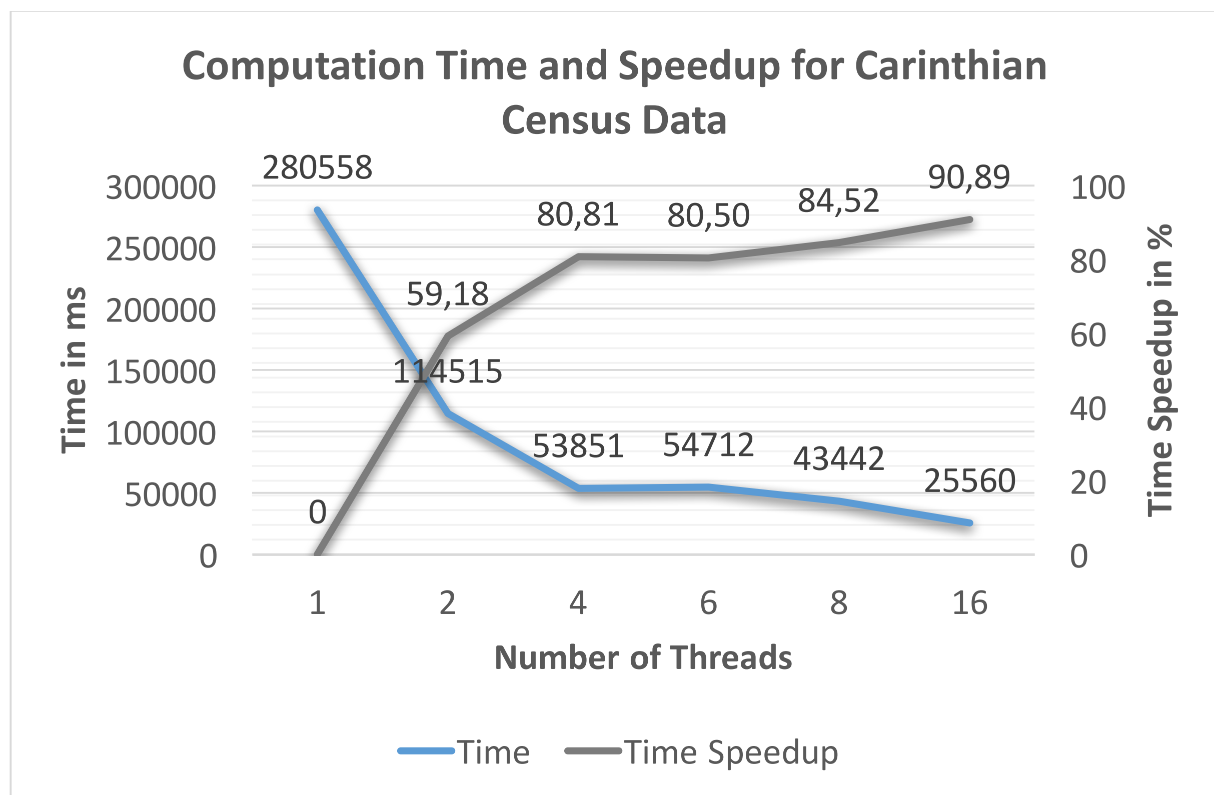

Figure 43 Computation time and speedup for Carinthian census data

The following Figure 44 shows the computation times as well as the relative changes in the AQE for a different number of threads. The blue line shows again the computation times. The orange line shows the AQE. In general, it is increasing apart from an outlier at four threads. When the AQE for one thread is used as a reference value, the AQE increases about ten percent at eight threads and about 21 percent at 16 threads. In absolute values, all AQEs are three digits behind the comma but nevertheless this shows that the AQE is increasing with a rise of the number of threads using the same parameters and input data.

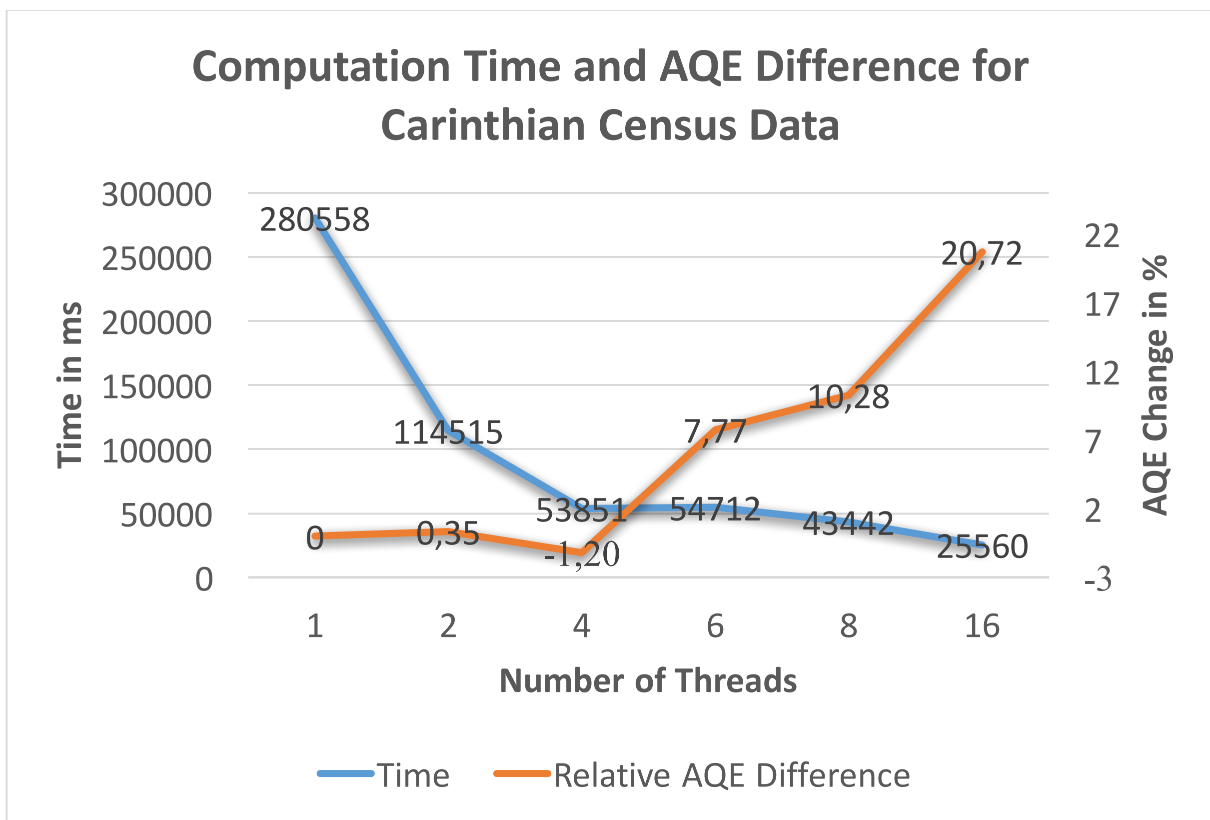

Figure 44 Computation time and AQE difference for Carinthian census data

All computations with the second dataset, the multispectral data, were run using the server at CUAS. The applied parameters are listed at the end of the prior chapter 6.2.2. To get a more detailed result considering the computation time, this value was split up into two values for the training: the training time itself and the total time, which contains also the computation of the AQE and the writing to the three output files. The time for the BMU search after training was split up into three values: the BMU search time, the AQE search time and the time for the writing of the "bmus.geojson" file.

When testing the time and error for the multispectral data, one, four, eight and sixteen threads were used ten times each. To use the BMU projection after training and compare the results, the same multispectral input data was split up into two parts: one part containing one fifth (18191 vectors) and the other part the remaining four fifth (72764 vectors) of the original dataset.

These two datasets where created programmatically via Java and have an evenly distributed content. The bigger dataset was trained using SOMatic with the following parameters: neurons on x = 270, neurons on y = 270, alpha value = 0.04, radius = 270, iterations in first run = 10000, similarity measure = 2, number of threads = 4, rounding = true, scaling factor = 2, training runs = 3, iterations in second run = 20000 and iterations in third run = 50000. After the training, the smaller dataset was projected onto the output SOM of the training using the parallelized BMU search. All computations were run on the server.

The following diagram shows the computation time split up into the training plus the computation of the AQE and the writing as well as the speedup for the bigger part of the multispectral data. The training time is significantly decreasing with a higher number of threads, which corresponds to an increase in the speedup. Compared to one thread, which represents a serial computation, the speedup increases almost 66 when using four, 79 when using eight and 87 percent when using sixteen threads. There is almost no change in the AQE plus writing time due to the fact that these values are mostly reliable on the size of the input data. The total time that the SOMatic Trainer took to create the output files and compute the AQE are then the blue and green values combined for each thread. This means for one thread, the mean runtime was 1069851 milliseconds, or almost 18 minutes.

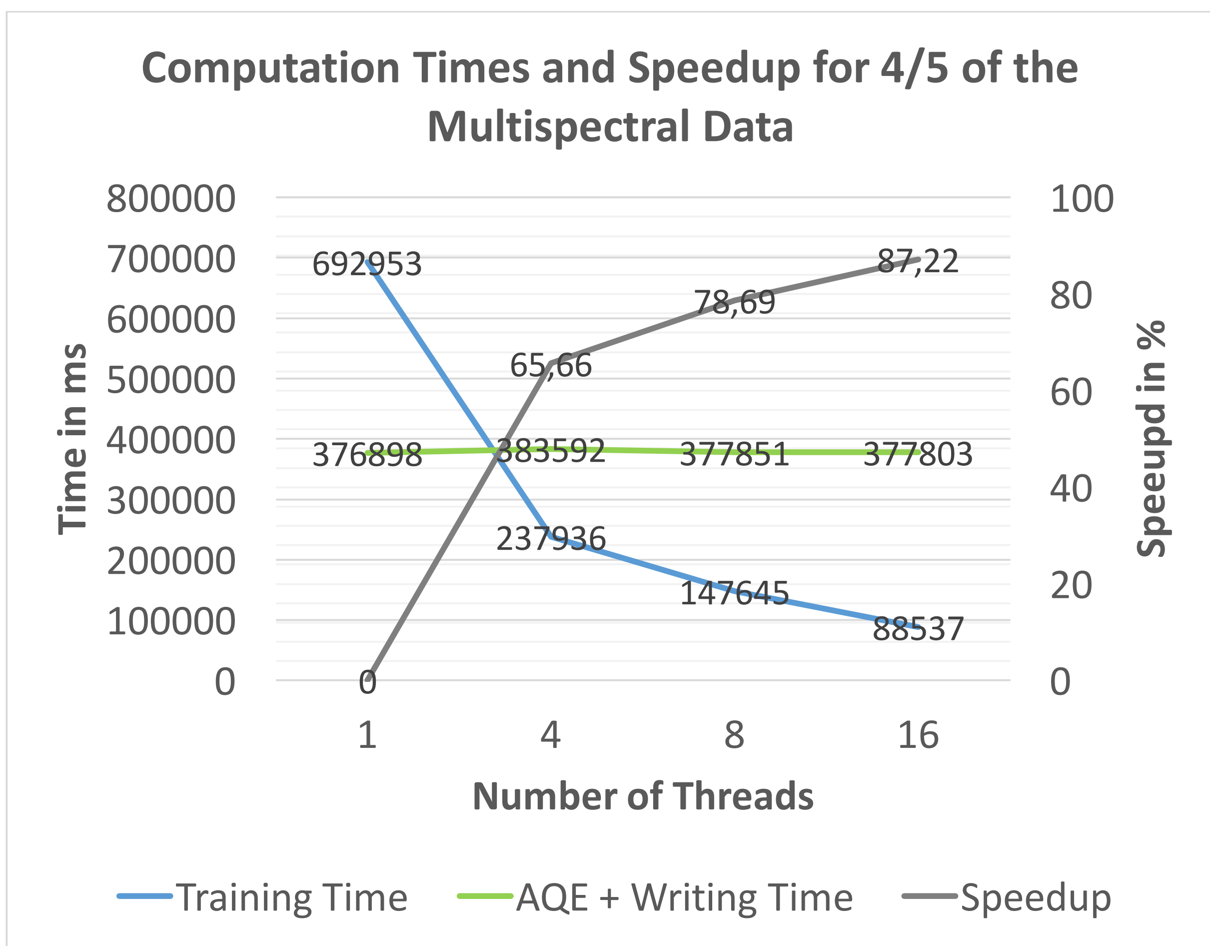

Figure 45 Computation times and speedup for bigger part of the multispectral data

The second value that differs with the thread number is the AQE. Figure 46 visualizes the relative AQE difference compared to the usage of one thread as well as the training time again as a reference. In relative values, the rise of the AQE is quite significant with an increase of 166.58 percent when using four and up to 242.46 percent when using sixteen threads. This means that the output SOM when using 16 threads is almost two and a half times worse than the output when using a serial computation according to the AQE. In absolute values, the differences are three or two digits behind the comma. Compared to the results for the Carinthia census data, this is a much higher decrease in quality, which can partially be explained by using the factor of ten less training steps for each of the three training runs. Nevertheless, it is showing that the quality of the SOM is decreasing when the number of threads is increasing.

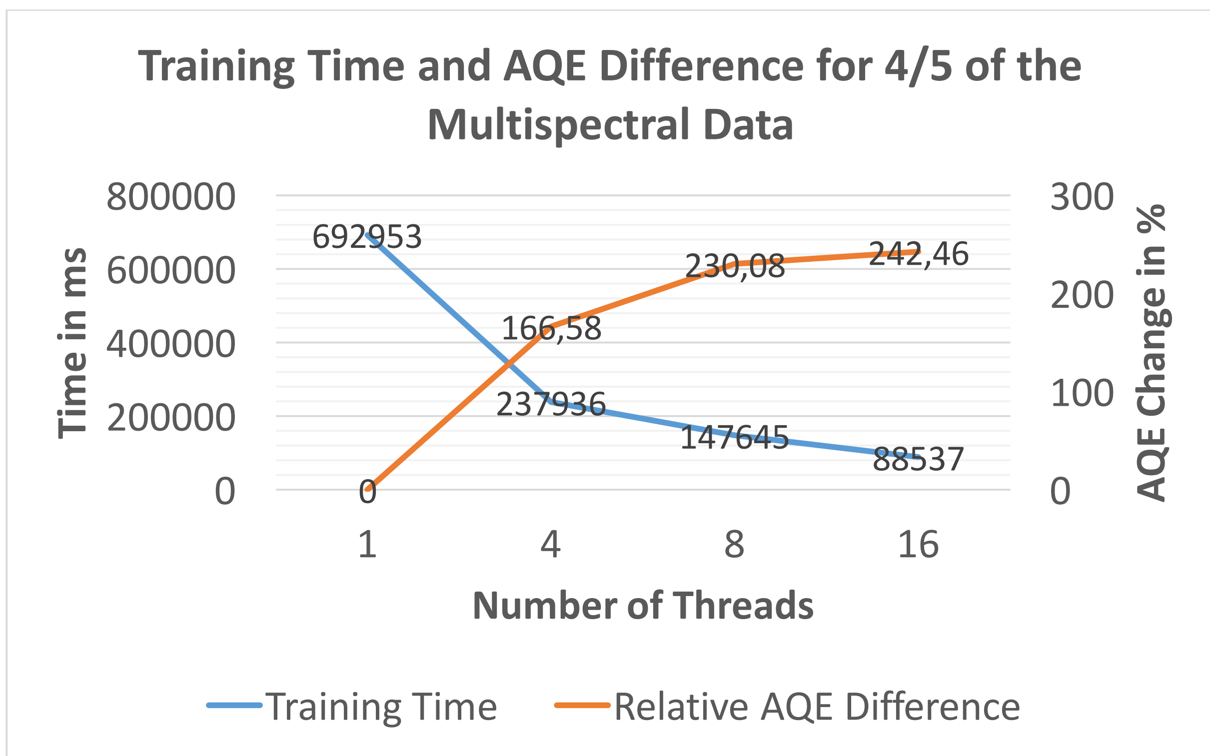

Figure 46 Training time and AQE difference for bigger part of multispectral data

The next figures look at the time and quality values for the projection of the smaller part of the multispectral data onto the SOM created by the bigger part. The first one shows the median values for the four different thread numbers for the BMU search time and the speedup of this computation step. As a reminder, this is only the time for the BMU search, so the computation time for the AQE and the writing of the output file is not included in this value. This diagram is using median values because the mean would get distorted by some values, which were way different from most of the others. This could be due to some activity on the server at the background. The BMU computation time can be significantly decreased via increasing the number of threads. Via applying a parallelization on 16 threads, the search time could be reduced by about 86 percent. The difference between eight and 16 threads is quite small because of the already low times of around ten and a bit under nine seconds. As already mentioned it also takes time to setup a thread pool and start the threads from there, meaning that the time differences decrease if the ratio between data size (18191 vectors in this case) and number of threads decreases as well. This implies that the grey curve will presumably go down and the blue curve go up again if the level of parallelization would be further increased here.

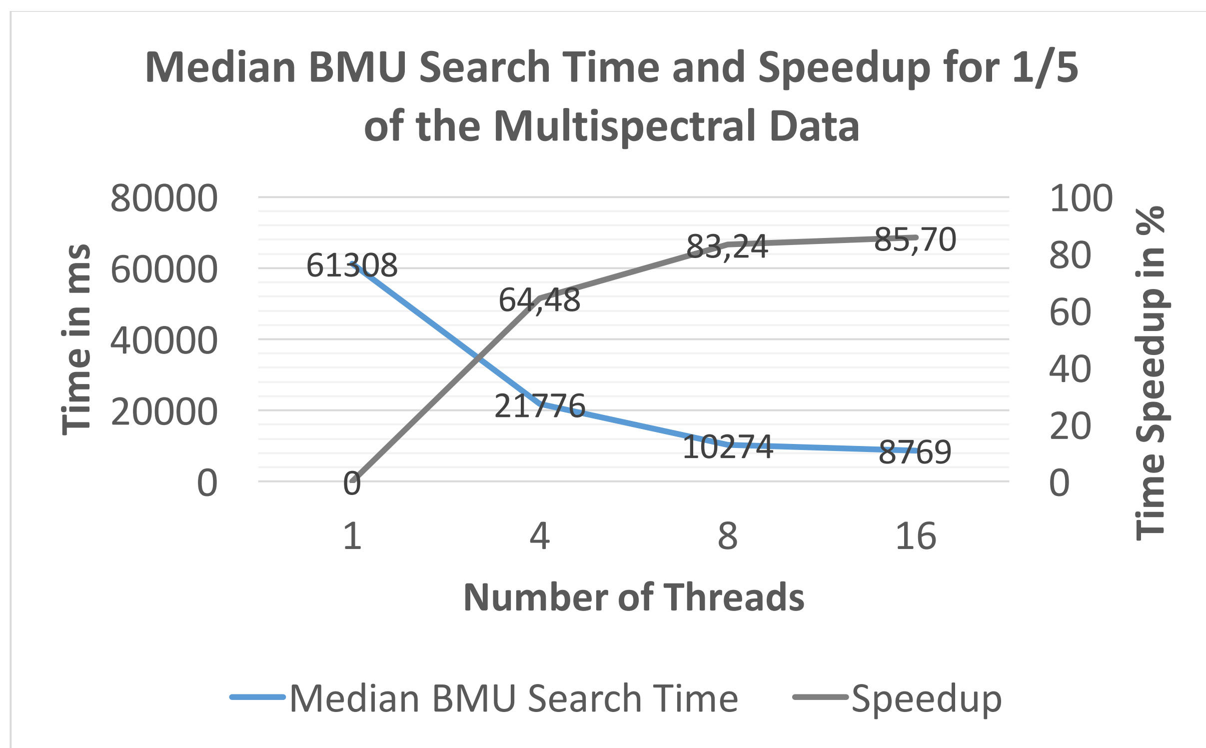

Figure 47 Median BMU search time and speedup for smaller part of multispectral data

When looking at the orange line showing the AQE change in the BMU computations displayed in Figure 48, one can see that it grows with a higher number of threads. When compared to the changes in Figure 46 though, an increase in parallelization here does not harm the quality of the result of the projection too much. In fact, the quality shrinks between a serial computation and using four threads by about one fourth, but does not decrease much more when increasing the thread number to eight or sixteen.

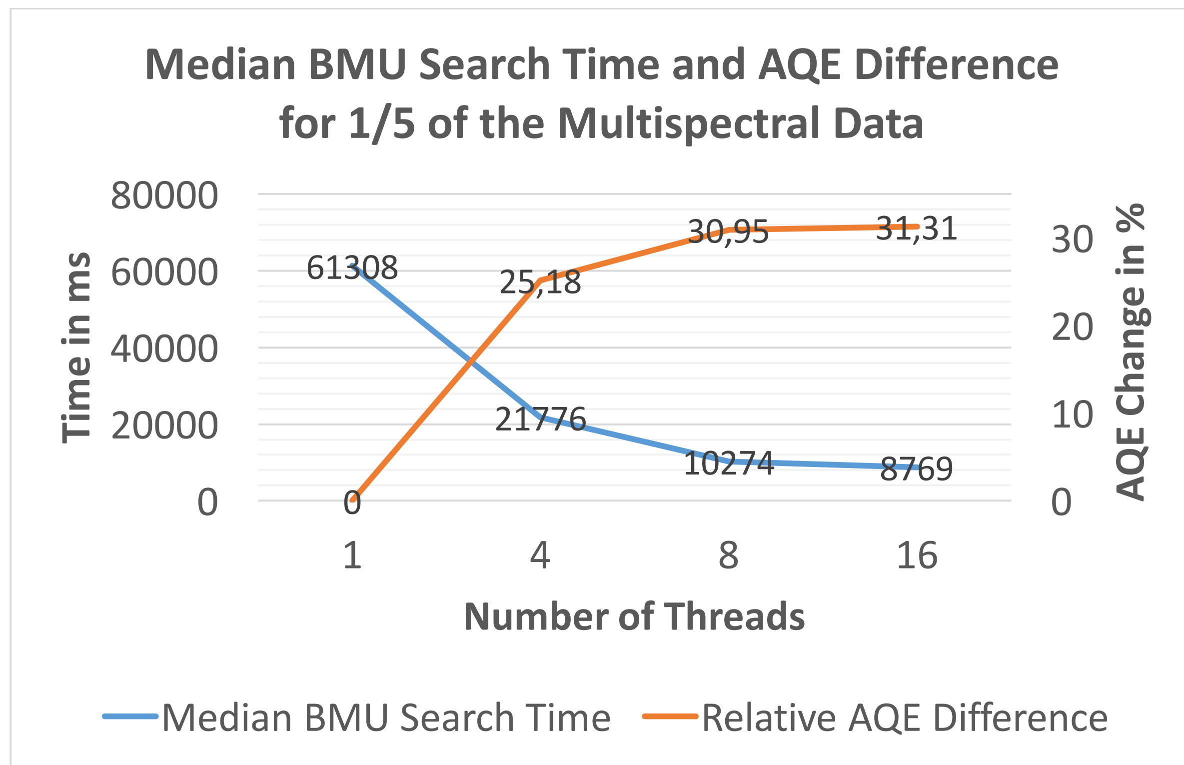

Figure 48 Median BMU search time and AQE difference for smaller part of multispectral data

The computation times for the AQE as well as the writing of the output files are not visualized specifically because they did not have a big fluctuation range within the test runs, apart from one outlier, which has as already mentioned presumably something to do with background computations on the server.

6.3 Cluster Computation

It was planned to have a cluster computation project that could run the SOMatic Trainer on a simple basis on different network clusters having the prerequisites setup and running. Actually, there is no implemented result in the cluster computation as it could not be finished as explained in 5.5 and 5.6. The output here are the architecture and workflows described in 4.4.2 and through continuing with the explained implementation steps, it should be achievable to get the SOMatic Trainer up and running in a JPPF cluster computing environment.

7. Discussion and Conclusion

The first research question how the output of SOMatic can be more useful for further usage (in the GIS world) can be answered with the integration of the GeoJSON component. The SOMatic trainer is in respect to the literature review for this project the first SOM implementation, which is able to import and export GeoJSON. There exists another one that can use JSON as input and output (Zasso, 2016) but not the geographic version of it. To integrate that functionality was of high importance within this implementation as GeoJSON is a standardized format by the Internet Engineering Task Force (IETF) and easy to integrate in GI-applications as well as web applications and other software. The XML format could have served as another solution but it lacks the geographic part. With GeoJSON it is easy to store and extract geographic features like points, lines, or polygons. The objects can be visualized in a GIS or web interface together with their respective attributes quite easily. As the background of the project member and the advisors lies in the GIS world, the decision to use GeoJSON instead of any other data store format was made with ease. The visualizations of the census data show that this data format is suited as an additional output format for the SOMatic Trainer.

To be able to apply the integration of the GeoJSON output directly into a GIS or web interface the geometry is stored as polygons. This increases the size of the GeoJSON by a significant amount, as for each neuron not only one coordinate pair has to be stored (the mid-point) but seven (six corner points plus the first one again so the polygon is closed). The geometry for the "bmus.geojson" file is even bigger because it stores not only the whole output SOM as polygons, but also the point coordinates for each input vector in its respective BMU. The purpose of this implemented structure is for the output to be usable immediately without further processing or computations. This is the reason why the neurons of the SOM are stored as polygons and not as points or why the input vectors also have coordinates assigned to them and not just the ID of the corresponding BMU.

The second and third research questions refer to how SOMatic could be integrated into a cluster computing network and what the threshold is beyond which it is better to train SOMs with SOMatic within a cluster network than on one machine These two cannot be answered to full satisfaction, or not at all. The cluster computation functionality is worked out in theory and described via diagrams and two scenarios. The first scenario is rather for testing purposes and does not necessarily lead to an increase in the quality of the SOM as without a data partitioning, the slaves can eventually find different BMUs for the same data vector. This could lead to a total different evolution of the attribute vectors of the neurons and therefore, the output SOM itself can also look differently.

Via the usage of a data partition method, the second scenario is certainly applicable since it has already been described in several best practice examples (Skupin, et al., 2013) (Yang & Ahuja, 1999). Unexpected events and a prolonged implementation of the prior programming steps lead to the fact that the cluster computation could not be finished within this project. The JPPF framework turned out to be more complex and the main issue is centered about the way the SOMatic trainer is implemented with using a global class to have access to and control all parameters throughout the whole training process. To be able to function in a cluster environment, the objects need to be serializable on the one hand, and not dependent onto one class on the other hand. In the first version it was not intended to use the SOMatic Trainer in a distributed environment. JPPF can only handle serializable traffic within its architecture. Sending objects like a SOM in Java was thus not possible. More robust software and compatibility testing in earlier stages of the project or more time at the end would have been possible solutions to be able to develop a strategy to work around the problems that came up considering the SOMatic Trainer and JPPF.

The fourth research question about the right balance factor between computation time and quality of the output SOM is answered with the diagrams in the section 6.2.3. Depending on the data size and available time, one can argue about a higher or lower level of parallelization during the training of the SOM. What comes out undoubtedly is the clear time decrease when applying parallelization. The speedup for all three test scenarios were about 90 percent when using 16 threads compared to a serial computation, which is a significant decrease in computation time. When looking at the quality, the AQE can serve as a good indicator for the correctness of the output SOM. The results show that its value is in a direct proportion to the number of threads. When looking at the census data, the increase was far lower than compared to the test runs of the multispectral data (20 compared to 242 percent). This would imply that a higher amount of training runs has a diminishing effect on the increase of the AQE over a higher number of threads.

As more training runs cause a more similar output SOM compared to the input data and the AQE computes the average difference between them, this diminishing effect should definitely be considered when opting for a good balance factor between quality and time. The quality should therefore be able to uphold when increasing the number of training runs with the number of threads at the same time. More quality measuring tools though would be helpful for further analyzation of the correctness of a SOM. Furthermore, not all possible thread numbers were tested. The intermediate values for the non-visualized thread numbers are therefore unknown, but would presumably lie in between the computed values. When it comes to projecting new data onto an existing SOM via the BMU search after training, the results show that the quality, visualized through the AQE, suffers less from an increase in parallelization.

The integration of the BMU search after training into SOMatic 2.0 is helpful as it allows to visualize data that was not part of the training. It can be very useful in a working deployment that consists of large SOMs to be able to track and monitor movement (changes) in a hyperspectral space. The projection of 25 percent of the original data size used to train the SOM as shown in Figure 48 produces a relative increase of the AQE of 31.31 percent when a serial computation is compared to a parallel one with using sixteen threads. This increase is way lower than the rise of the error in training, which lies at 242.46 percent for sixteen threads. It shows that applying this technique does not cause a big loss in quality when compared to the quality decrease if the thread numbers are increased in the training. This of course can only be stated as true for the applied size of the test data. If the ratio between the data to project and the trained data is too high, or the vectors are too different, the quality would supposedly suffer more.

The SOMatic Trainer 2.0 is exploiting the parallelization functionality using the conventional SOM training on one machine to its maximum, as also the BMU projection is integrated and parallelized. The batch training on the other hand, described in section 2.2.2, is not used in the 2.0 version as it was not part of the goals in this project. The advantages of this SOM training computation though, especially when looking at the simpler parallelization of the batch processes are definitely not to underestimate. An integration of this computation form would not have been possible within the scope of this project but should definitely be considered in the implementation of a SOMatic Trainer 2.1 or 3.0. It is not sure, if a parallelized batch training is faster than the parallelization as it is implemented in 2.0 but what can definitely not occur in a batch training is the issue with two or more threads trying to update the same neuron at the same time if the updates are then performed sequentially. If they should also be performed in parallel, this possible latency could occur as well but as this weight update is only performed once per batch, it might not be needed to parallelize the update.

Within this project, the programming language number one was Java as the SOMatic Trainer, as well as JPPF are written in this language. This is rather different to other SOM implementations, where hardware-closer programming languages like C++ (Wittek, et al., 2016) are used. The main reason why this one is written in Java is the fact, that the knowledge of the people who use it and the people who wrote and extended it is highest in the Java world. The argument, that hardware-closer programming languages might be faster performance wise can be countered by the exploited speed-ups, which can be achieved by using different parallelization techniques.

The starting point was not to create diagrams for a new application to program but an already existing piece of software that meant to be updated and extended. That is quite different to a task where you have to start from scratch and has some advantages but also disadvantages. One could say that it is easier, because you have already a working program but that also comes with a limitation on the technologies that you would like to apply and integrate in your project.

8. Future Work

In a possible SOMatic 3.0 implementation, it would be very convenient to add to the conventional algorithm the batch version (described in section 2.2.2) because of two main reasons: it is faster and also easier to parallelize. The user of the library could then choose which computation method she or he wants to use and could maybe also compare the results of them. Another part of a future extension would be to restructure the class structure and parameter usage within the SOMatic Trainer as with the current usage of the class Global.java, it is not possible to integrate it directly into a cluster computation system. The parallelization on one machine is with an integration of the batch computation already heavily exploited. Therefore, to achieve a further speed up and be able to process more data, the structure has to be rethought and changed. This could be achieved for example with a more object oriented approach, which would allow to have several SOM objects and therefore computations within the same Java code.

The split up of the computation times in the results show that the computation of the AQE is directly proportional to the data size. The AQE could easily be parallelized as well through applying a data partition approach in its implementation. Each thread could compute the sum of the differences between the input vectors and their corresponding BMUs in attribute space for its part of the data and these values would then be summed up and averaged at the end giving the final AQE for the whole SOM. As it only computes the difference in attribute space, this computation is also embarrassingly parallel. That means in this case the full parallelization of a local machine could be used and the output will not suffer any quality loss through an increased parallelization. Alongside with the AQE, a second or third technique to rate the quality of an output SOM like the topographic product or the topographic error (Pölzlbauer, 2004) would be of interest as well. What was not tested in the evaluation of this project were the effects of different parameters onto the quality using two or more data sets of the same type. With the integration of more quality measurements, the results when testing different parameters would probably lead to a better understanding of which parameter setting fits which data size the most.

Another parallelization technique for the BMU search, which came up through the literature search and was thought through in the conceptual phase but not implemented afterwards is described in Figure 49. The neurons of the SOM can be split up evenly on the threads that will be used in the BMU search. Each thread can then compute the BMU for its set of neurons and the winner neuron is found by computing the minimum of these local BMUs. It was not implemented because as of the current way parallelization works in the SOMatic Trainer (through dividing the training steps as described in section 3.3.1) each thread would then need to create at least one other thread to be able to distribute the neurons among these two or more threads. It would be of interest to implement that parallelization technique in a batch version of the SOM algorithm and compare it with a data partitioning approach in respect to which of the two methods is faster or slower and more accurate. The following figure shows the workflow to implement a parallel BMU search in the training process of the SOMatic trainer for one input vector using a SOM with 150 neurons.

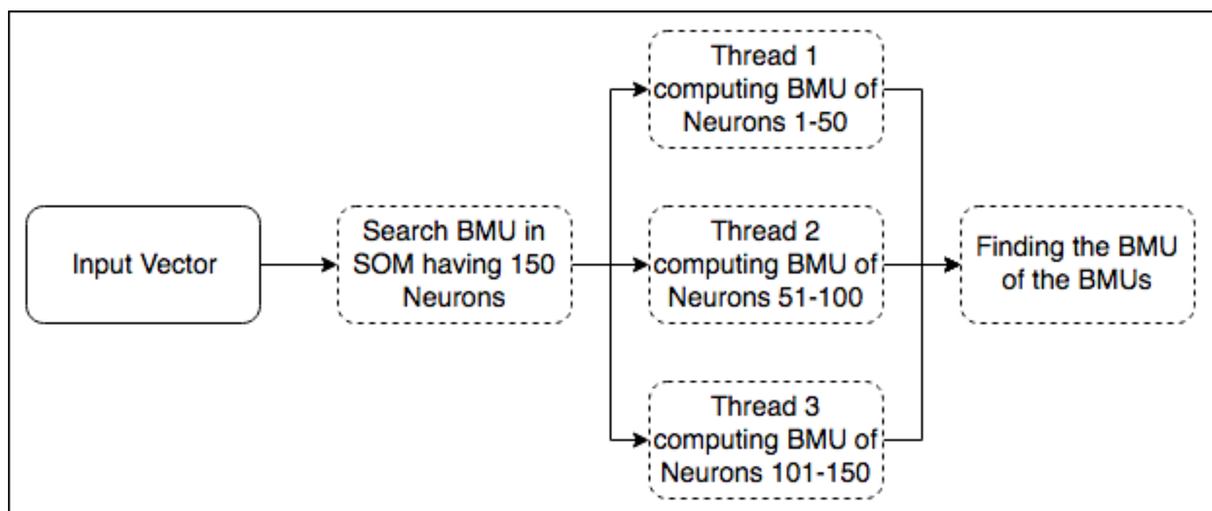

Figure 49 Workflow of parallel BMU search in the training process of SOMatic

The current GeoJSON SOM output of the SOMatic Trainer is always stored as polygons. If the user of that software does not need polygons but still wants to make use of the GeoJSON output, it would be nice to store the neurons only as points to minimize the data size. One step further would be to provide three options: a GeoJSON output with polygons, a GeoJSON output with points and a pure JSON output without any geometry. This would help to enlarge the possible fields of application for the SOMatic JSON output.

References

Apache Foundation, 2017. *Apache Hadoop.* [Online] Available at: http://hadoop.apache.org/ [Accessed 02 April 2017].

Bandeira, N., Lobo, V. J. & Moura-Pires, F., 1998. *Training a Self-Organizing Map distributed on a PVM network.* Lisbon, IEEE Conference Publications, pp. 457 - 461.

Bell, N. & Hoberock, J., 2011. Thrust: A productivity-oriented li- brary for cuda. In: *GPU Computing Gems: Jade Edition.* Morgan Kaufmann, pp. 359-373.

Bullinaria, J. A., 2004. *Self Organizing Maps: Algorithms and Applications.*

Butler, H. et al., 2016. *The GeoJSON Format.*

Dagum, L. & Menon, R., 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, January, 5(1), pp. 46-55.

Dean, J. & Ghemawat, S., 2004. *MapReduce: Simplified Data Processing on Large Clusters.* San Francisco, USENIX Association, pp. 137-150.

Fort, J.-C., Patrick, L. & Marie, C., 2002. *Advantages and drawbacks of the Batch Kohonen algorithm.* Bruges, Belgium, ESANN 2002, pp. 223-230.

Garcia, C., Prieto, M. & Pascual-Montano, A., 2006. *A Speculative Parallel Algorithm for Self-Organizing Maps.* Jülich, John von Neumann Institute for Computing, pp. 615-622.

Hawick, K. A., Coddington, P. D. & James, H. A., 2003. Distributed frameworks and parallel algorithms for processing large-scale geographic data. *Parallel Computing*, October, 29(10), pp. 1297–1333.

Jordan, J. & Angelopoulou, E., 2013. *hyperspectral image visualization with a 3D self-organizing map,* Erlangen.

JPPF.org, 2016. *JPPF v5.2 Manual.*

JPPF.org, 2017. *A first taste of JPPF.* [Online] Available at: http://www.jppf.org/doc/5.2/index.php?title=A_first_taste_of_JPPF [Accessed 13 06 2017].

Kohonen, T., 1990. *The Self-Organizing Map.* IEEE.

Kohonen, T., 1993. *Things you haven't heard about the Self-Organizing Map.* Piscataway, NJ, IEEE, pp. 1147-1156.

Kohonen, T., 1998. The self-organizing map. *Neurocomputing*, Volume 21, pp. 1-6.

Kohonen, T., 2013. Essentials of the self-organizing map. In: Elsevier, ed. *Neural Networks 37.* Elsevier, pp. 52-65.

Kohonen, T., Hynninen, J., Kangas, J. & Laaksonen, J., 1996. *SOM-PAK: The self-organizing map program package,* Helsinki.

Lawrence, R. D., Almasi, G. S. & Rushmeier, H. E., 1999. A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems. *Data Mining and Knowledge Discovery*, Volume 3, pp. 171–195.

Li, Q., Kecman, V. & Salman, R., 2010. *A chunking method for Euclidean distance matrix calculation on large dataset using multi-GPU.* Washington, DC, pp. 208-213.

Minhoe, K., Souhwan, J. & Minho, P., 2015. *A Distributed Self-Organizing Map for DoS Attack Detection.* Soongsil University Seoul.

NVidia Corporation, 2014. *Compute Unified Device Architecture Programming Guide 6.0.*

Open GIS Consortium Inc., 1999. *OpenGIS Simple Feature Specification For SQL.* Open GIS Consortium (OGC).

Open MPI Project Team, 2017. *Open MPI.* [Online] Available at: https://www.open-mpi.org/ [Accessed 02 April 2017].

Pölzlbauer, G., 2004. *Survey and Comparison of Quality Measures for Self-Organizing Maps,* Vienna.

Rainer, M., 2013. *SOMatic Viewer: Implementation of an interactive self-organizing map visualization toolset in Processing and Java.* Villach: Carinthia University of Applied Sciences.

Silva, B. & Marques, N., 2007. *A Hybrid Parallel SOM Algorithm for Large Maps in Data-Mining.* Guimaraes, IEEE Guimaraes.

Skupin, A., Biberstine, J. R. & Börner, K., 2013. Visualizing the Topical Structure of the Medical Sciences: A Self-Organizing Map Approach. *PLoS ONE*, March, 8(3).

Snir, M. et al., 1996. *MPI–The Complete Reference: The MPI Core.* Cambridge(Massachusetts): The MIT Press.

Spöcklberger, M., 2013. *SOMatic Trainer: Implementation of a Self-Organizing Map Tool with Parallel Training for Processing applied to Carinthian Municipality Census Data.* Villach: Carinthia University of Applied Sciences.

Sul, S.-J. & Tovchigrechko, A., 2011. Parallelizing BLAST and SOM algorithms with MapReduce-MPI library. *IEEE International Parallel & Distributed Processing Symposium*, 16-20 May, pp. 476-484.

Tole, A. A., 2013. Big Data Challenges. *Database Systems Journal*, 03, Volume IV, pp. 31-40.

van de Sande, K. E. A., Gevers, T. & Snoek, C. G. M., 2011. Empowering visual categorization with the GPU. *IEEE Transactions on Multimedia*, February, 13(1), pp. 60-70.

Wehrens, R. & Buydens, L. M. C., 2007. Self- and super-organizing maps in R: The kohonen package. *Journal of Statistical Software*, October, 21(5), pp. 1-19.

White, T., 2015. *Hadoop: The definitive Guide.* 4th Edition ed. Sebastopol(CA): O'Reilly Media Inc..

Wikimedia Foundation Inc., 2017. *Open MPI.* [Online] Available at: https://en.wikipedia.org/wiki/Open_MPI [Accessed 02 April 2017].

Wikimedia Foundation Inc., 2017. *Wikipedia: GeoJSON.* [Online] Available at: https://en.wikipedia.org/wiki/GeoJSON [Accessed 31 03 2017].

Wikimedia Foundation Inc., 2017. *Wikipedia: Java (programming language).* [Online] Available at: https://en.wikipedia.org/wiki/Java_(programming_language) [Accessed 19 02 2017].

Wittek, P. & Darányi, S., 2012. *A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment.* Bruges, pp. 609-614.

Wittek, P., Gao, S. C., Lim, I. S. & Zhao, L., 2016. *Somoclu: An Efficient Parallel Library for Self-Organizing Maps.* University of Boras, ICFO - The Institute of Photonic Sciences, Tsinghua University, Bangor University: s.n.

Yang, M.-H. & Ahuja, N., 1999. *A Data Partition Method for Parallel Self-Organizing Map.* Washington DC, pp. 1929-1933.

Zasso, M., 2016. *github.* [Online] Available at: https://github.com/mljs/som [Accessed 18 07 2017].

List of Figures

Figure 1 Hexagonal SOM display in coordinate system with different alignment (Spöcklberger, 2013) 4
Figure 2 SOM created out of words in medical publications (Skupin, et al., 2013) 10
Figure 3 Explanation of one training iteration of the SOM algorithm 11
Figure 4 Pseudo-code representing the conventional SOM algorithm (Lawrence, et al., 1999) 12
Figure 5 Pseudo-code representation of the batch SOM algorithm (Lawrence, et al., 1999) 13
Figure 6 Starting state (left) and 2D comparison of the results of the on-line (middle) and the batch computation (right) (Fort, et al., 2002) 14
Figure 7 Starting state (left) and 3D comparison of the results of the on-line (middle) and the batch computation (right) (Fort, et al., 2002) 14
Figure 8 Java "Hello World" example code 15
Figure 9 GeoJSON example 17
Figure 10 Example of the flow of a MapReduce operation (Dean & Ghemawat, 2004) 19
Figure 11 Overview of a JPPF network topology (JPPF.org, 2016) 20
Figure 12 Job-Workflow in the JPPF topology (JPPF.org, 2016) 21
Figure 13 Execution times of data-partition vs. hybrid method (Silva & Marques, 2007) 23
Figure 14 Master-slave architecture in a speculative parallel SOM algorithm (Garcia, et al., 2006) 24
Figure 15 Comparison of raw image, conventional SOM and parallel SOM (Yang & Ahuja, 1999) 26
Figure 16 Curves of computation time (Bandeira, et al., 1998) 28
Figure 17 Execution time comparison between GPU and CPU (Wittek & Darányi, 2012) 29
Figure 18 Overview of a master-slave architecture 31
Figure 19 Runtime comparison on a 50x50 sized SOM (Wittek, et al., 2016) 31
Figure 20 Runtime comparison on a 200x200 sized SOM (Wittek, et al., 2016) 32
Figure 21 Parameters needed to run the SOMatic trainer 33
Figure 22 Example of a .sprj output file 34
Figure 23 Sequence of the training threads within the SOMatic trainer (Spöcklberger, 2013) 35
Figure 24 One training step parallelized onto two threads in SOMatic 35
Figure 25 Neighborhood overlapping (purple neurons) of two BMUs (red and dark blue) in parallel training 36
Figure 26 Project workflow overview 38
Figure 27 Example for Data parallelization approach in the BMU search 41
Figure 28 Scenario one of the distributed computation of SOMatic 42
Figure 29 Scenario two of the distributed computation of SOMatic 43
Figure 30 Workflow of the JPPF Implementation of the SOMatic Trainer 44
Figure 31 Example of a library dependency in a Maven POM.xml file 45
Figure 32 Java code example showing the writing of the polygon coordinates into JSON Arrays 47
Figure 33 Hexagonal neuron alignment example with applied numeration 48
Figure 34 Characteristics of a 30-60-90 degree triangle 49
Figure 35 Usage of the 30-60-90 degree triangle characteristics within a hexagon 50
Figure 36 Code-excerpt of the parallelized BMU search after training 51
Figure 37 GeoJSON example (Butler, et al., 2016) 53
Figure 38 Class diagram of the SOMatic Trainer 2.0 55
Figure 39 Carinthian census data: BMU GeoJSON visualization from QGIS 58
Figure 40 Carinthian census data: Distribution of ratio of Austrian citizens 59
Figure 41 Carinthian census data: Distribution of ratio of non-EU citizens 59
Figure 42 True color images from the Batiquitos Lagoon in the North of SD showing this region in April 1993 (left), 2003 (middle) and 2013 (right) 60

Figure 43 Computation time and speedup for Carinthian census data 63
Figure 44 Computation time and AQE difference for Carinthian census data 64
Figure 45 Computation times and speedup for bigger part of the multispectral data 66
Figure 46 Training time and AQE difference for bigger part of multispectral data 67
Figure 47 Median BMU search time and speedup for smaller part of multispectral data 68
Figure 48 Median BMU search time and AQE difference for smaller part of multispectral data 69
Figure 49 Workflow of parallel BMU search in the training process of SOMatic 74

List of Tables

Table 1 Execution times for different map sizes and numbers of PCs (Bandeira, et al., 1998) 28
Table 2 Mean of the computation time (in ms) and AQE for ten computation runs using the Carinthia census data 62