DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Domain Parallelization of SGDE based Classification

Jan Schopohl

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Domain Parallelization of SGDE based Classification

Domänen-Parallelisierung von SGDE basierter Klassifikation

Author:	Jan Schopohl
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor:	Kilian Röhner, M.Sc.
Submission Date:	15.04.2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.04.2019 Jan Schopohl

Abstract

This thesis describes cluster-level domain-parallelization of a sparse grid density estimation (SGDE) based classification algorithm. The Online phase of the implementation of this algorithm in the SG++ toolbox was parallelized using MPI and the ScaLAPACK library. This means that distributed compute resources can be efficiently used as the implementation is not constrained to one node. Instead of a data-parallel approach, model parallelization was chosen in order to achieve greater flexibility and efficiency on complex grids. The parallel classifier was integrated into the existing datamining pipeline of the SG++ toolbox. The performance of the new approach was evaluated for weak and strong scaling and the influence of parallelization parameters was considered. The results of the scaling tests were mostly as expected, however refinement was identified as a bottleneck.

Zusammenfassung

Diese Arbeit beschreibt die Domänen-Parallelisierung eines Klassifikationsalgorithmus der auf Dünngitter Dichteschätzung basiert. Die Online Phase des Algorithmus wurde mit Hilfe von MPI und der ScaLAPACK Bibliothek auf Cluster-Ebene parallelisiert und in die SG++ Bibliothek integriert. Dadurch können Rechenressourcen effizient genutzt werden und der Algorithmus ist nicht auf einen Rechnerknoten beschränkt. Statt eines Datenparallelen Ansatzes wurde eine Parallelisierung des Modells ausgewählt, mit dem Ziel größere Flexibilität und Effizienz bei komplexen Gittern zu erreichen. Der parallelisierte Algorithmus wurde in die bestehende Datamining Pipeline der SG++ Bibliothek integriert. Die Auswirkungen von verschiedenen Parametern auf die Leistung wurden untersucht und es wurden Tests zur Skalierbarkeit durchgeführt. Die Ergebnisse dieser Tests waren weitgehend wie erwartet, allerdings stellt die Verfeinerung des Gitters einen Engpass dar.

Contents

Abstract **iii**

1. Introduction **1**

2. Theoretical Background **3**
2.1. Machine Learning 3
2.2. Sparse Grids 5
2.3. Sparse Grid Density Estimation and Classification 8

3. Parallelized Algorithm **11**
3.1. Evaluation of Parallelization Concepts for SGDE 11
3.1.1. Node-Level and Cluster-Level Architectures 11
3.1.2. Parallelization Approaches 12
3.2. Current State 13
3.3. Proposed Algorithm 15
3.4. Matrix Distribution 17

4. Implementation **20**
4.1. ScaLAPACK 20
4.2. Matrix distribution 21
4.3. Integration of the Parallel Algorithm into the Datamining Pipeline . . . 23

5. Evaluation **27**
5.1. Analysis of the proposed algorithm 27
5.2. Scalability 29
5.2.1. Strong Scaling 29
5.2.2. Weak Scaling 33

6. Future Work **37**

7. Conclusion **39**

A. Appendix **40**

Bibliography **45**

1. Introduction

As the means to acquire and process large amounts of data have rapidly increased, the capabilities of data mining and machine learning algorithms are steadily growing. There is a wide range of applications for these methods, such as autonomous driving and analysis of medical images. In machine learning, one typically wants to learn patterns from large sets of data. While the increasing quantity of data helps to create more accurate models, it also means that a large amount of computing power is needed. This is especially the case for large and complex models, which are often needed to achieve a good solution.

As a result, a single compute node, even with efficient node-level parallelization, might not have sufficient power to train and evaluate the model in a reasonable time frame. Therefore, parallelization has to be done on a cluster-level to efficiently use all available resources.

A common machine learning task is classification, which means that a model should be trained to map from a data sample to a class label. Algorithms that solve this problem for high dimensional data commonly have to use dimensionality reduction in order to keep the method computationally feasible. However, this can cause a reduction of classification performance, as important information in the data can be lost.

One approach to solve this problem is by using classification based on sparse grid density estimation (SGDE). This algorithm first estimates the probabilistic density using basis functions centered on grid points in the data space and then classifies the data according to the highest probability given by the density functions. To keep the dimensionality of the data and avoid information loss, sparse grids are used instead of full grids: The grid adapts to the problem space and only keeps grid points in areas of high interest, for example along the decision boundary. This enables a balance between model accuracy and complexity. A more detailed description of these concepts can be found in Chapter 2.

The full implementation of these structures is quite complex, however the SG++ library provides implementations for a wide range of sparse grid algorithms [Pfl10].

As large and complex models are needed needed in order to achieve good performance and generalization, and the datasets can be very large, there is a need to efficiently parallelize the SGDE algorithm in SG++. In the past, domain parallelization has mainly been done on a node level using frameworks such as OpenMP [HP11].

For data parallelization, there have been efforts for cluster-level parallelization of this method [Bod17], however the model itself was not distributed. As the use of domain-parallelization might lead to more flexible and more efficient use of computing resources, this approach is desirable. Therefore, this thesis describes the concepts, algorithmic changes and implementation of this form of cluster-level parallelization for the SGDE-based classification algorithm in SG++. The algorithmic considerations are described in Chapter 3. The new algorithm was integrated into the datamining pipeline of the SG++ library, this is described in Chapter 4. Additionally, the performance and scaling of this approach are evaluated in Chapter 5.

2. Theoretical Background

This chapter gives a basic introduction to the concepts of Machine Learning and Sparse Grids. Furthermore, the principle of the SGDE algorithm and Offline/Online splitting are explained.

2.1. Machine Learning

The basic concept of Machine Learning is to automatically extract useful patterns from data [Mur12; Bis06]. The common goal is to train a model using training data and achieve good generalization on unseen data. The first step is to train the model using a training set $\mathcal{S}_{train}$. The output of the model can then be evaluated using an error metric. A common problem is that a model memorizes the exact training data and produces significantly worse results on unseen data. This is referred to as *overfitting* and means that the model does not generalize well. To identify this problem, a model should always be evaluated on a different dataset than the training set. Therefore it is common to split the whole dataset $\mathcal{S}$ into training set $\mathcal{S}_{train}$, validation set $\mathcal{S}_{val}$ and test set $\mathcal{S}_{test}$. The validation set can be used to analyse the performance of the model during training and subsequently tweak the model parameters. As this can implicitly also cause overfitting on the validation set, the final evaluation of the model should be done on a previously unseen test set, as this will show whether the model has generalized properly.

When approximating a function, large oscillations in the predicted value for small variations of the input are usually the result of overfitting. One approach to avoid this problem is *regularization*, which prevents such irregularities by enforcing smoothness criteria. Regularization operators are usually denoted as Λ, while the strength of regularization can be controlled by a factor λ. A common example is to penalize large coefficients in the model.

While there are many different tasks in machine learning, we can generally identify two main categories: *Supervised learning* and *unsupervised learning*.

Supervised Learning The goal of algorithms of this type is to approximate an unknown function $f(\mathbf{x}) = y$, so mapping from an input $\mathbf{x}_i$ to a target y_i. In the case that y is a discrete class label $y \in \{1, \dots, C\}$, we call this a *classification* problem. In this case, the goal is to find a good prediction $\hat{y}$ for the unknown function using a set of training data $\mathcal{S}_{train} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$, which is made up from M samples of d-dimensional input $\mathbf{x}_i$ and corresponding labels y_i.

Unsupervised Learning In unsupervised learning, there are no labels given for training, just M d-dimensional data samples $\mathbf{x}$. Hence the training set in this case is just $\mathcal{S}_{train} = \{\mathbf{x}_i\}_{i=1}^{M}$, which also means that there are multiple possible learning tasks. One common task is *density estimation*, where the model tries to learn the probability density function $p(\mathbf{x})$ of the data set.

SGDE based classification In the case of SGDE based classification, the supervised learning task is based on density estimation for every class, so a unsupervised machine learning method. The basic approach is to learn the class conditionals $p(\mathbf{x}|y = c)$ using sparse grid density estimation. This can be done by splitting the training set into samples of each class, namely sets $\mathcal{S}_{c_{train}} = \{\mathbf{x}_i \in \mathcal{S}_{train} : y_i = c\}$ [Peh13]. Then we can train models that estimate $p_c(\mathbf{x}) = p(\mathbf{x}|y = c)$ on these training sets. The prior class probabilities $p(y = c)$ can be estimated as empirical probabilities $p(y = c) = \frac{|S_{c_{train}}|}{|S_{train}|}$ and the posterior $p(y = c|\mathbf{x})$ can then be computed using Bayes' rule:

$$p(y = c|\mathbf{x}) = \frac{p(\mathbf{x}|y = c)p(y = c)}{\sum_{c'=1}^{C} p(\mathbf{x}|y = c')p(y = c')} \tag{2.1}$$

Note that for classification, we are only interested in the maximum posterior, so we can ignore the denominator in Bayes' rule, as it is just a normalizer and the same for every class. Thus, the classifier can be defined as

$$\hat{c}(\mathbf{x}) = \underset{c' \in \{1,\dots,C\}}{\arg\max}\, p(\mathbf{x}|y = c)p(y = c) = \underset{c' \in \{1,\dots,C\}}{\arg\max}\, p(y = c|\mathbf{x}) \tag{2.2}$$

This is considered a probabilistic, generative approach to classification, as it does not directly learn a mapping between input data and class labels but instead uses class conditional probabilities and Bayes' rule to derive the estimate.

2.2. Sparse Grids

Consider a grid-based approach to estimate the probability density function on a data set with d-dimensions. In the case of a full grid and refinement level n, we discretize the domain using equidistant grid points with a mesh width of $h_n = 2^{-n}$ in each dimension [Pfl10]. This means that we have to evaluate $\mathcal{O}(h_n^{-d}) = \mathcal{O}(2^{dn})$ grid points for interpolation of the function. This exponential growth when adding dimensions is referred to as *curse of dimensionality* and makes a full-grid approach computationally infeasible for high dimensional problems. Therefore, sparse grids are used, which try to adapt to the data so only relevant subspaces of the grid have to be evaluated. The following sections introduce the basic concepts of sparse grids, which are largely the same as for full grids. For a more detailed review of sparse grids, refer to [BG04; PPB10; Peh13].

Hierarchical basis functions Grid-based approaches rely on a set of hierarchical basis functions, here the simple hat function will be used:

$$\varphi(x) := \max(1 - |x|, 0) \tag{2.3}$$

It should be noted that there are multiple other basis functions that can be used, details about these can be found in [Pfl10]. Sparse-grids are constructed using a hierarchy of levels, meaning the basis functions are translated and dilated to be centered on grid points $x_{l,i} := 2^{-l}i$, where l is the level and i is an index $0 < i < 2^l$:

$$\varphi_{l,i}(x) := \varphi(2^l x - i) \tag{2.4}$$

One important property of this hierarchy is that the support of basis functions is getting smaller with increasing level l, which means that the influence of these basis functions is getting more limited to local parts of the interpolated function. This concept can be seen in Figure 2.1. As we are interested in the d-dimensional case, we extend the basis functions using a tensor product,

$$\varphi_{\boldsymbol{l},\boldsymbol{i}}(\mathbf{x}) = \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j), \tag{2.5}$$

where $\boldsymbol{l}$ and $\boldsymbol{i}$ now are d-dimensional indices for level and index in each dimension. As a note on notation, sparse grid basis functions are sometimes referred to with only one index instead of $\boldsymbol{l}$ and $\boldsymbol{i}$, this still means that a hierarchy is used, the grid points are just enumerated for simplicity.

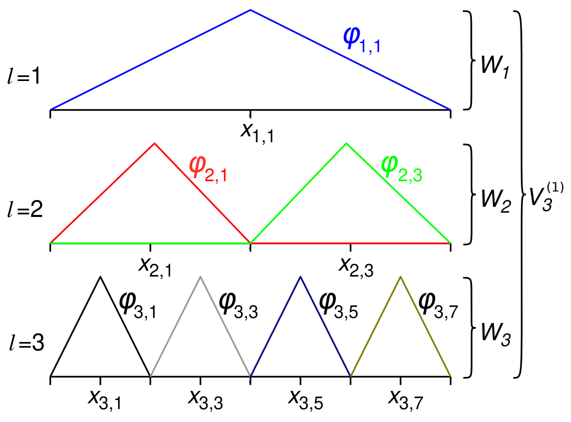


Figure 2.1.: One-dimensional hierarchical basis functions which form the (sparse) grid space $V_3^{(1)}$. Figure adapted from [Pfl10].

To define the subspaces $W_{\boldsymbol{l}}$ for each level and dimension of the grid, index sets are introduced,

$$I_{\boldsymbol{l}} := \{\boldsymbol{i} \in \mathbb{N}^d : 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ odd}, 1 \leq j \leq d\}, \tag{2.6}$$

then we can define the hierarchical subspaces as

$$W_{\boldsymbol{l}} := \text{span}\{\varphi_{\boldsymbol{l},\boldsymbol{i}}(\mathbf{x}) : \boldsymbol{i} \in I_{\boldsymbol{l}}\}. \tag{2.7}$$

To obtain a sparse grid space $V_n^{(1)}$, we sum up a selection of these subspaces,

$$V_n^{(1)} := \bigoplus_{|\boldsymbol{l}|_1 \leq n+d-1} W_{\boldsymbol{l}}, \tag{2.8}$$

using the l_1-norm $|\boldsymbol{l}|_1 = \sum_i l_i$. The two-dimensional sparse grid space with $l = 3$ and its subspaces are shown in Figure 2.2. We can then define the interpolation $\hat{f}(\mathbf{x}) \in V_n^{(1)}$ on a sparse grid as the sum of basis functions, each multiplied with a hierarchical surplus $\alpha_{\boldsymbol{l},\boldsymbol{i}}$:

$$\hat{f}(\mathbf{x}) := \sum_{|\boldsymbol{l}|_1 \leq n+d-1, \boldsymbol{i} \in I_{\boldsymbol{l}}} \alpha_{\boldsymbol{l},\boldsymbol{i}} \varphi_{\boldsymbol{l},\boldsymbol{i}(\mathbf{x})}. \tag{2.9}$$

The idea behind this is that a sparse grid essentially selects the subspaces with the highest contribution to the interpolation to maintain accuracy while reducing complexity. As the number of grid points is reduced, the cost for evaluation of the interpolation is reduced to $\mathcal{O}(h_n^{-1}(\log_2 h_n^{-1})^{d-1}) = \mathcal{O}(2^n n^{d-1})$, which is a significant improvement compared to the cost for a full grid of $\mathcal{O}(2^{dn})$.

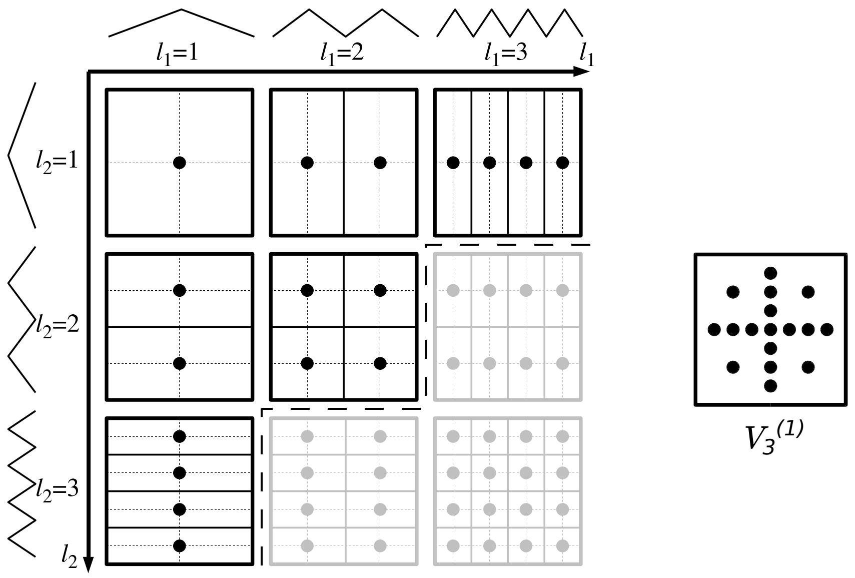


Figure 2.2.: Hierarchical subspaces for the two-dimensional case of a sparse grid space $V_3^{(1)}$. Next to the axis, the basis functions are shown for each level. On the right, a sparse grid formed from all subspaces in $V_3^{(1)}$ is shown. Figure taken from [Pfl10].

Adaptivity Depending on the interpolation problem, some areas of the grid space might require a finer grid than others. Therefore the sparse grid can be refined and coarsened in certain areas or dimensions in order to better adapt to the problem, which means achieving a better solution without increasing costs too much. As evaluating all possible refinement candidates is infeasible, an adaptivity criterion is needed [Pfl10; Peh13]. A simple criterion is to refine the grid points with the highest absolute surplusses $|\alpha_{l,i}|$, as this indicates a large influence on the interpolation. Because of this, it makes sense to add the hierarchical children of such points. This criterion is widely used and quite robust, however it is important to keep in mind that it does not fit every application.

2.3. Sparse Grid Density Estimation and Classification

While it is possible to directly estimate a classification function using a sparse grid, a method introduced by [Peh13] can naturally cope with multiple classes and usually leads to better results. The method builds a probabilistic classifier based on density estimation for every class, such as introduced in Section 2.1. This means that an efficient density estimation method is needed. Using a training data set $S_{c_{train}} = \{\mathbf{x}_i\}_{i=1}^{M}$, an initial guess p_ϵ can be created: As an example, let $p_\epsilon = \frac{1}{M}\sum_{i=1}^{M} \delta_{\mathbf{x}_i}$, where $\delta_{\mathbf{x}_i}$ is a Dirac delta function centered on the training point $\mathbf{x}_i$. This initial value is overfitted on the training set, however the density function can then be estimated by adding regularization with spline smoothing and searching for a function in the space $\mathcal{V}$ such that

$$\hat{p}(\mathbf{x}) = \underset{f \in \mathcal{V}}{\arg\min} \int_\Omega (f(\mathbf{x}) - p_\epsilon(\mathbf{x}))^2 \mathrm{d}\mathbf{x} + \lambda ||\Lambda f||_{L^2}^2, \tag{2.10}$$

where regularization strength can be controlled by the parameter λ. As further described in [Peh13], this can then be transformed into a system of linear equations:

$$(\boldsymbol{R} + \lambda \boldsymbol{C})\boldsymbol{\alpha} = \boldsymbol{b}. \tag{2.11}$$

Let N be the number of grid points and $\langle \cdot, \cdot \rangle_{L_2}$ the L_2 inner product. $\boldsymbol{R}$ is a $N \times N$ matrix with entries $R_{ij} = \langle \varphi_i, \varphi_j \rangle_{L_2}$, where the entries essentially describe the overlap of the basis functions. $\boldsymbol{C}$ is a $N \times N$ matrix as well and the entries are defined as $C_{ij} = \langle \Lambda\varphi_i, \Lambda\varphi_j \rangle_{L_2}$, where Λ is an regularization operator. It is often set to the identity matrix, so $\boldsymbol{C} = \boldsymbol{I}$, as this provides efficient regularization. Only the right hand side of the linear system is dependent on the data, as vector $\boldsymbol{b}$ of size N is defined as $b_i = \frac{1}{M}\sum_{j=1}^{M} \varphi_i(\mathbf{x}_j)$. The system can then be solved in order to calculate the vector of surplusses $\boldsymbol{\alpha}$ and interpolate the density function as $\hat{p}(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \varphi_i(\mathbf{x})$. A major advantage of this method is that the $N \times N$ matrix on the left hand side, $(\boldsymbol{R} + \lambda \boldsymbol{C})$, which is referred to as *system matrix*, is not dependent on the data set. This means that a factorization of the system matrix can be pre-computed in an *Offline phase*, and the linear system is then solved using the training data in an *Online phase*. As a result, the Online phase has a runtime complexity of $\mathcal{O}(N^2)$ instead of $\mathcal{O}(N^3)$ without an Offline/Online split [Peh13]. This property is very useful, as the same factorization of the system matrix can often be used multiple times, for example to train with multiple batches or data, on different datasets or to evaluate hyperparameter settings.

Offline phase The Offline phase is used to compute a factorization of the system matrix $(\boldsymbol{R} + \lambda \boldsymbol{C})$. This means that the grid parameters and regularization method have to be chosen in this phase. The system matrix is then decomposed in order to allow efficient solve operations of the linear system in the Online phase. Multiple different decomposition types have been implemented and evaluated in the SG++ toolbox before. However, currently mostly the Cholesky or Orthogonal decomposition is used, as both allow for refinement of the grid and therefore updates of the decomposition during the Online phase. Another criterion for the choice of decomposition type is the ability to vary regularization strength during the Online phase. This is currently only fully supported by Eigen decomposition and Orthogonal decomposition. A comparison of different decomposition types is provided by [Fuc18], the Cholesky decomposition and Orthogonal decomposition are also described in the following paragraphs.

Online phase In the Online phase, vector $\boldsymbol{b}$ is computed with batches of training data of size M. Then the decomposition created in the Offline phase is used to solve the linear system for $\boldsymbol{\alpha}$. After this training step, the performance of the model can be evaluated on a validation set and, depending on decomposition type, the grid might be refined and the regularization strength tweaked. This algorithm can be repeated until sufficient performance is achieved. This thesis targets the parallelization of the Online phase of the model, as this step is usually repeated many times and therefore runtime is an important property.

Cholesky decomposition The Cholesky decomposition was integrated into the SGDE learner by [Sie16]. During the Offline phase, the symmetric and positive definite system matrix is decomposed as

$$\boldsymbol{R} + \lambda \boldsymbol{C} = \boldsymbol{L}\boldsymbol{L}^T, \tag{2.12}$$

where $\boldsymbol{L}$ is a lower triangular matrix. In the Online phase, we can then solve the linear system $(\boldsymbol{L}\boldsymbol{L}^T)\boldsymbol{\alpha} = \boldsymbol{b}$ using forward and back substitution. The factorization can be efficiently updated after grid refinement or coarsening. The regularization strength λ can also be altered, however this has a runtime complexity of $\mathcal{O}(N^3)$, meaning it is of the same order as recomputing the whole factorization.

Orthogonal decomposition To efficiently update the regularization strength, an Orthogonal decomposition can be used, which was implemented for SGDE by [Bos17]. In this case, the Offline phase consists of decomposing the matrix $\boldsymbol{R}$ into

$$\boldsymbol{R} = \boldsymbol{Q}\boldsymbol{T}\boldsymbol{Q}^T, \tag{2.13}$$

where $\boldsymbol{Q}$ is orthogonal and $\boldsymbol{T}$ is a symmetric tridiagonal matrix. Setting $\boldsymbol{C} = \boldsymbol{I}$, the linear system can then be solved in the online phase using basic linear algebra operations:

$$\boldsymbol{\alpha} = \boldsymbol{Q}\boldsymbol{T}^{-1}\boldsymbol{Q}^T\boldsymbol{b} + \boldsymbol{B}\boldsymbol{b}. \tag{2.14}$$

The matrix $\boldsymbol{B}$ is an additive component which is needed for updates of the decomposition after refinement. While this method is overall slightly slower than the Cholesky decomposition, the grid can be adapted and regularization strength can also be changed efficiently.

3. Parallelized Algorithm

This chapter describes the choice of the parallelization type and explains the concept of the parallelized algorithm.

3.1. Evaluation of Parallelization Concepts for SGDE

There is a number of different parallelization types that can be applied to the SGDE algorithm. When looking at the architecture of the parallel system, we can differentiate between *node-level* and *cluster-level* approaches. Looking at how the algorithm decomposes the problem, one can further distinguish *data-parallel* and *model-parallel* approaches, where the *domain* of the model is parallelized.

3.1.1. Node-Level and Cluster-Level Architectures

The most common form of parallelization can be found on a node-level, using a shared-address-space for communication. This can be done using frameworks such as OpenMP, which offers convenient compiler directives to automatically parallelize parts of a program. In the SG++ library, this has already been used to parallelize evaluations of functions, for example using the UpDown scheme [Pfl10]. An alternative approach was used by [HP11], where vectorization techniques were combined with OpenMP parallelization and lead to better performance. OpenMP has also been used to parallelize parts of the SGDE algorithm [Let17]. However, all these concepts are limited to a shared-memory system, so usually one node. This avoids communication costs, but also introduces a synchronization overhead, as it has to be ensured that shared data is not corrupted, so mechanisms like locks are needed. Furthermore, the resources of clusters cannot be exploited using node-level parallelism.

We can solve this problem by using cluster-level parallelization with a message passing architecture. This means that multiple nodes, each with their own memory and address space, can be used. Communication between the nodes is handled by sending and receiving messages, which is commonly done using the message passing interface (MPI) protocol. This increases communication overhead, however it can avoid the need for locks and other synchronization techniques, as the address spaces are not shared, so data corruption cannot occur as easily. The most obvious advantage of this architecture is the ability to harness the resources of large clusters. It should also be noted that MPI programs can run on a single node if needed, as an environment with separate address spaces and message passing can easily be emulated. Therefore the message passing architecture can offer a high degree of scalability. In SG++, parallelization with MPI has previously been implemented using batch parallelization, which is elaborated in the next section [Bod17]. This thesis proposes another cluster-level parallelization approach. A more detailed comparison of parallel architectures can be found in [Kum02].

3.1.2. Parallelization Approaches

When looking beyond architectural differences in parallelization, an important choice is the decomposition of the problem. One approach is to decompose the data and operating on different parts of the data in parallel, which is referred to as *data parallelism*. Another approach is to decompose the model domain itself, this is called *model* or *domain parallelism*.

Data parallelization When using data parallelization, a common approach is to split the dataset into batches. These batches can then be mapped onto different worker nodes and the model can be trained on multiple nodes in parallel, each using a different data batch. The results can then be combined on a master node. This approach was proposed and implemented by [Bod17]. As all nodes train the whole model, each node has to possess a copy of it and keep it synchronized with the other nodes. This leads to the advantage that batches can be distributed onto the nodes efficiently using a scheduling algorithm and the load can be balanced well on the cluster. However, this also creates a communication overhead, as batch assignments, results and model synchronization messages have to be sent.

Furthermore, the current implementation is not integrated into the SG++ datamining pipeline, where SGDE-based classification was implemented by [Fuc18]. This structure streamlined and integrated the codebase of different datamining methods in SG++ and offers a user friendly interface. An integration of the parallel batch learner into the pipeline was evaluated, however this was not possible with reasonable effort, as it would require major changes in the datamining pipeline.

Domain Parallelization In order to integrate a cluster-level parallelization approach into the datamining pipeline, a method using domain parallelization was chosen. This means that the model itself is decomposed and mapped onto different nodes, where every node trains part of the model, each using the same data.

Combigrid parallelization Two different approaches for this were evaluated, *combigrid* and *spatial* decomposition. The combigrid approach uses the sparse grid combination technique, this means solutions for subspaces of the sparse grid, which are full grids themselves, can be calculated in parallel and then combined to a sparse grid solution. Details can be found in [Pfl10]. The advantage of such a method is that solvers for full grids can be used on the subspaces. While this sort of split of a sparse grid is straightforward to implement and would minimize communication overhead, the overall cost for the computation of the solution using this method is higher. Furthermore, spatial adaptivity of the grid is not possible when using full subspaces, which severely limits the usefulness of this method for larger SGDE problems.

Spatial parallelization Therefore a spatial decomposition of the model was chosen as parallelization model. The concept of this is to split the grid space, which for SGDE can be achieved by splitting the linear system $(\boldsymbol{R} + \lambda \boldsymbol{I})\boldsymbol{\alpha} = \boldsymbol{b}$ and distributing it across multiple nodes. This approach can be integrated into the datamining pipeline with reasonable effort and full adaptivity of the grid is possible. However, the linear system has to be synchronized for refinements and potentially be redistributed, which adds a communication overhead. Compared to the combigrid approach, it does not carry an additional computational overhead, as a direct sparse grid solver is used. Another advantage of this approach is that the granularity of the parallelization can be controlled by the type of split of the system matrix. This makes the approach highly configurable and adaptable to different system architectures.

3.2. Current State

The current state of the classification algorithm in the SG++ datamining pipeline is shown in Algorithm 1. The latest version was implemented by [Fuc18] and uses a classification method based on SGDE, as described in Section 2.3. This means that a SGDE model is trained for every class. Note that the algorithm is a batch learner and processes only part of the dataset in each iteration. This helps to reduce the memory needed and the computational cost per iteration to compute vector $\boldsymbol{b}$. Additionally, it offers support for streaming, so adding new data to the model while training. As a consequence, the computation of $\boldsymbol{b}_t$ for the current iteration t also incorporates the vector $\boldsymbol{b}_{t-1}$ from the previous iteration:

$$\boldsymbol{b}_t = \frac{1}{M_{total}}(\beta \boldsymbol{b}_{t-1} + \sum_{j=1}^{M} \varphi(\mathbf{x}_j)) \tag{3.1}$$

Algorithm 1 Current classification algorithm that is implemented in the datamining pipeline.

```
1: Load dataset, split training and validation data
2: for epoch $e \leftarrow 1 \dots E$ do
3:     Reset datasource (shuffle if necessary)
4:     while datasource has another batch of training data do
5:         $\mathcal{S}_{train} \leftarrow$ datasource.nextBatch()
6:         Split the batch into classes to obtain subsets $S_{c_{trian}}$
7:         for $c \leftarrow 1 \dots C$ do
8:             compute $\boldsymbol{\alpha}_c$: UPDATE_SGDE($S_{c_{trian}}$)
9:         Evaluate the model on the validation set and calculate the error
10:        if refinement necessary (can be based on error) then
11:            Refine or coarsen the grid
12:            Update the factorization and size of datastructures
13:
14: function UPDATE_SGDE($\mathcal{S}_{c_{train}}$)
15:     Input is $\mathcal{S}_{c_{train}}$ with $M$ samples of class $c$
16:     if first iteration then
17:         Create grid
18:         Load system matrix factorization
19:         Initialize datastructures needed for Online phase
20:                                          ▷ compute the density function
21:     Use the $M$ samples in the batch to calculate $\boldsymbol{b}_t = \frac{1}{M_{\text{total}}}(\beta \boldsymbol{b}_{t-1} + \sum_{j=1}^{M} \varphi(\mathbf{x}_j))$
22:     solve the linear system $(\boldsymbol{R} + \lambda \boldsymbol{C})\boldsymbol{\alpha} = \boldsymbol{b}$ using the factorization
```

The factor β weights the previous batches and is usually set to 1, M_{total} refers to the total number of points in all batches up to and including the current. The step of actually solving the linear system varies depending on the factorization that was used in the Offline phase.

After the SGDE models have been updated, grid refinement or coarsening may take place. There are numerous configuration options for this step: Usually, refinement takes place based on the error on the validation set. How many refinement steps are considered and how many points are refined in each step can also be configured, along with the criterion according to which points are chosen for refinement. After each refinement, the factorization of the system matrix has to be updated, which is different depending on factorization type.

The next section explains how the model of this algorithm can be parallelized.

3.3. Proposed Algorithm

As discussed in Section 3.1.2, the goal of this thesis is to split up the model and distribute it. This means splitting the grid, which can also be achieved by distributing the linear system $(\boldsymbol{R} + \lambda \boldsymbol{I})\boldsymbol{\alpha} = \boldsymbol{b}$, as each row can be seen as representing a basis function of the grid.

In Algorithm 2, the proposed algorithm is shown, changes compared to the current algorithm are highlighted in red. An important point is that the main structure of the algorithm can remain similar, which makes it easier to integrate it into the datamining pipeline. The most important difference is the distribution of the system matrix factorization and the distributed generation of $\boldsymbol{b}_t$ along with the solve operation. The exact distribution of the matrices and vectors is described in the next section.

Furthermore, there is no clear master and worker distinction of tasks, instead all tasks perform almost the same steps, except for a few minor details. The reason for this is that as a result of the matrix distribution, static scheduling is used, as matrix operations can be distributed quite evenly and dynamic scheduling would cause constant redistribution of the matrices and vectors. Additionally, the framework used for implementation, ScaLAPACK, is not suited for dynamic scheduling of operations.

As a result, refinement is also done by all nodes. This might seem like a waste of resources at first, but because of the static scheduling all processes have to stay synchronized quite closely , so all processes would have to wait if one process does the refinement. Additionally, there would be need for further network traffic and synchronization if the refinement would only be performed on one node, and the algorithm would get more complex, making it harder to integrate into the current structure of the pipeline. To improve the proposed solution, it is recommended that refinement is fully parallelized in the future, as discussed in Chapter 6.

When computing $\boldsymbol{b}$ or the error function, datapoints have to be evaluated on the sparse grid, meaning the interpolation of the function that the grid approximates has to be computed, which was introduced in Equation (2.9). While there are approaches which vectorize this operation by taking advantage of modern hardware [HP11], here the classical recursive implementation is used. This evaluation algorithm takes advantage of the property that only a single product $\alpha_{\boldsymbol{l},\boldsymbol{i}}\varphi_{\boldsymbol{l},\boldsymbol{i}}(\mathbf{x})$ in one subspace, so of the same multiindex $\boldsymbol{l}$, can be unequal to zero. This means that only a fraction of these products have to be evaluated. As a consequence, every process also has to keep a full local copy of the vector $\boldsymbol{\alpha}$ to execute this evaluation algorithm, because it might need to access values of $\boldsymbol{\alpha}$ that are not in its local split. Along with refinement this is the reason that $\boldsymbol{\alpha}$ has to be gathered on all processes after the update of the SGDE model.

Algorithm 2 Proposed parallel classification algorithm. Changes compared to the current algorithm that was described in Algorithm 1 are highlighted in red.

```
1: Load dataset, split training and validation data
2: for epoch $e \leftarrow 1 \dots E$ do
3:     Reset datasource (shuffle if necessary)
4:     while datasource has another batch of training data do
5:         $\mathcal{S}_{train} \leftarrow$ datasource.nextBatch()
6:         Split the batch into classes to obtain subsets $S_{c_{trian}}$
7:         for $c \leftarrow 1 \dots C$ do
8:             Distributed computation of $\boldsymbol{\alpha}_c$: UPDATE_SGDE_PARALLEL($\mathcal{S}_{c_{trian}}$)
9:         Gather $\boldsymbol{\alpha}$ on all processes, needed for evaluation and refinement
10:        Distributed evaluation of the model on the validation set, compute error
11:        if refinement necessary (can be based on error) then
12:            Refine or coarsen the grid
13:            Update the factorization and size of datastructures
14:            Redistribute factorization and other datastructures
15: function UPDATE_SGDE_PARALLEL($\mathcal{S}_{c_{train}}$)
16:     Input $\mathcal{S}_{c_{train}}$ with $M$ samples of class $c$
17:     if first iteration then
18:         Create grid
19:         Load system matrix factorization
20:         Distribute system matrix factorization
21:         Initialize (distributed) datastructures needed for Online phase
22:                                                  ▷ compute the density function
23:     Distributed calculation of $\boldsymbol{b}_t = \frac{1}{M_{\text{total}}}(\beta \boldsymbol{b}_{t-1} + \sum_{j=1}^{M} \varphi(\mathbf{x}_j))$
24:     Solve the distributed linear system $(\boldsymbol{R} + \lambda \boldsymbol{C})\boldsymbol{\alpha} = \boldsymbol{b}$ using the factorization
```

Depending on the factorization type used in the Offline phase, the solver for the linear system and the update and redistribution after a refinement are different. The currently most useful factorization types, Orthogonal and Cholesky factorization, are considered in the following paragraphs.

Orthogonal Decomposition In this case there are three matrices which have to be distributed, $\boldsymbol{Q}, \boldsymbol{T}^{-1}$ and $\boldsymbol{B}$. However, the linear system can then be solved with simple linear algebra operations, as described in Equation (2.14). Therefore the solve operation can be distributed by using a parallelized BLAS library, such as PBLAS which is included in ScaLAPACK, for details see Chapter 4.

Cholesky Decomposition For this factorization, only the lower triangular matrix $\boldsymbol{L}$ has to be distributed, which means less overhead compared to the Orthogonal decomposition. To solve the linear system, forward and backward substitution have to be performed. These operations can also be parallelized using the ScaLAPACK library.

3.4. Matrix Distribution

A central component of the proposed algorithm is the decomposition scheme of matrices and vectors. As the ScaLAPACK library was used for implementation, a two-dimensional block cyclic distribution scheme was chosen [Bla+97].

Assume we have $P_r \cdot P_c$ processes which form a process grid with P_r rows and P_c columns. Note that this process grid should not be confused with the sparse grid itself, it is just an organization structure for multiple processes. Thus, each process in the grid has a row index p_r and a column index p_c. Now, the goal is to distribute a matrix $\boldsymbol{A}$ or vector $\boldsymbol{a}$ evenly on all processes in order to efficiently perform distributed LAPACK or BLAS operations. The Scalable Linear Algebra PACKage (ScaLAPACK) library solves this by using the two dimensional block cyclic distribution.

The scheme will be explained for the case of a matrix $\boldsymbol{A}$ with r rows and c columns, however a vector can be mapped in the same way, it will be handled as a matrix with one column. First, the matrix is divided into two-dimensional blocks of b_r rows and b_c columns. Then, the blocks are cyclically assigned to the processes, so block (m, n) is assigned to process $(m \mod P_r, n \mod P_c)$. To efficiently use the local memory hierarchy of every process, the all blocks of one process are stored contiguously. This means we now have multiple indices for each entry of the matrix: The global index (i, j), the index of the element in the corresponding block (x, y) and the local index in the memory of the process (x_p, y_p), which is calculated using the coordinate of the block in the local memory of the process (l, m). Given the global index (i, j), the following formulas can be derived to compute the other indices:

$$(p_r, p_c) = \left(\left\lfloor \frac{i}{b_r} \right\rfloor \mod P_r, \left\lfloor \frac{j}{b_c} \right\rfloor \mod P_c \right) \tag{3.2}$$

$$(l, m) = \left(\left\lfloor \frac{i}{P_r \cdot b_r} \right\rfloor, \left\lfloor \frac{j}{P_c \cdot b_c} \right\rfloor \right) \tag{3.3}$$

$$(x, y) = (i \bmod b_r, j \bmod b_c) \tag{3.4}$$

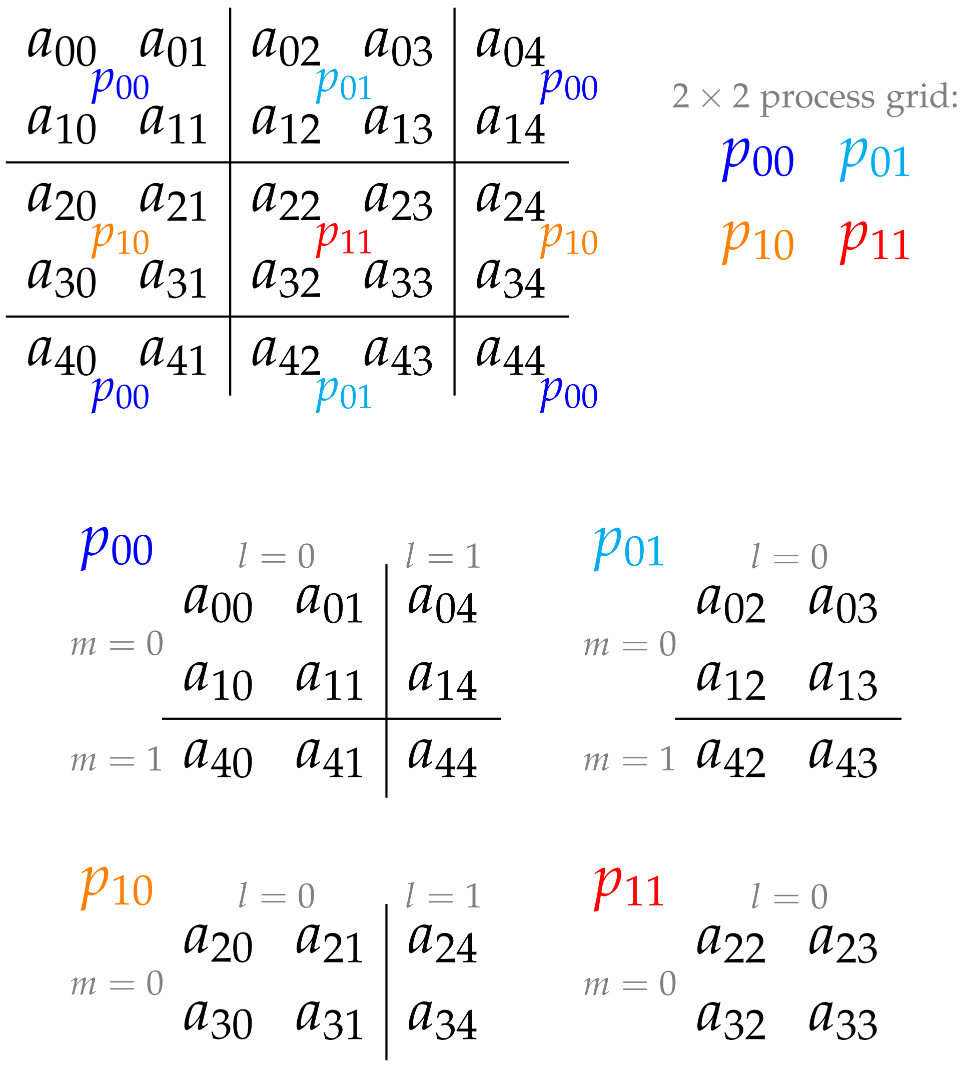
$$(x_p, y_p) = ((l \cdot b_r) + x, (m \cdot b_c) + y) \tag{3.5}$$

When given the local index (x_p, y_p) and the index of the process (p_r, p_c) of an element, the global index can be computed according to

$$(x_p, y_p) = \left(\left(\left\lfloor \frac{x_p}{b_r} \right\rfloor P_r + p_r \right) b_r + (x_p \bmod b_r), \left(\left\lfloor \frac{y_p}{b_c} \right\rfloor P_c + p_c \right) b_c + (y_p \bmod b_c) \right). \tag{3.6}$$

An example of a 5×5 matrix distributed onto a 2×2 process grid can be seen in Figure 3.1. While small matrices, such as in the example, are not necessarily distributed completely evenly, these differences even out for large matrices when an appropriate block size is chosen. Furthermore, a key advantage of the two-dimensional block cyclic distribution is that for large matrices, every process receives multiple parts of each row and column. This prevents bottlenecks in solve operations, as it assures that every process is part of most operations.

Another advantage of this scheme is the flexible configuration: The shape of the process grid and the block size can be changed to account for special cases. As the SGDE algorithm heavily depends on BLAS operations, we can change to process grid shape from a square to a row, so $p \times 1$ processes, to distribute the operations more efficiently, as further evaluated in Chapter 5. Additionally, the contiguous storage on every process enables local BLAS operations to take advantage of the memory hierarchy [Bla+97].



a_{00} a_{01} a_{02} a_{03} a_{04}
p_{00} p_{01} p_{00}
a_{10} a_{11} a_{12} a_{13} a_{14}
a_{20} a_{21} a_{22} a_{23} a_{24}
p_{10} p_{11} p_{10}
a_{30} a_{31} a_{32} a_{33} a_{34}
a_{40} a_{41} a_{42} a_{43} a_{44}
p_{00} p_{01} p_{00}
2×2 process grid:
p_{00} p_{01}
p_{10} p_{11}
p_{00} $l = 0$ $l = 1$ $m = 0$ $m = 1$
p_{01} $l = 0$ $m = 0$ $m = 1$
p_{10} $l = 0$ $l = 1$ $m = 0$
p_{11} $l = 0$ $m = 0$

Figure 3.1.: Example of a matrix of size $r = c = 5$ being mapped onto a process grid of size $P_r = P_c = 2$ according to the two-dimensional block cyclic distribution scheme. Block size is set to $b_r = b_c = 2$. In the upper part of the figure, the split of the global matrix into blocks is shown, along with the mapping of blocks onto processes. In the lower part, the local memory of the processes is shown.

4. Implementation

This chapter describes implementation of the parallel algorithm that was presented in the previous chapter in the SG++ library. In particular, it describes the integration of the ScaLAPACK library [Bla+97] into the SG++ codebase[1] and the changes made to the datamining pipeline. At the end of the chapter, challenges encountered during the implementation are described.

4.1. ScaLAPACK

The core of the parallel algorithm is the distributed solve operation of the linear system. As this operation is based on basic methods from linear algebra, there are libraries that provide optimized distributed implementations for this problem type. The Scalable Linear Algebra PACKage (ScaLAPACK)[2] library was chosen for this purpose. The advantage of this library is that is is widely available on high performance computing systems and has a similar interface as the Linear Algebra PACKage (LAPACK)[3] and the Basic Linear Algebra Subprograms (BLAS)[4]. However, most parts of ScaLAPACK are written in Fortran, which means that special care has to be taken when ScaLAPACK routines are called from C++. This was deemed an acceptable tradeoff for the advantages offered by ScaLAPACK.

The main part of ScaLAPACK provides distributed versions of common LAPACK functions. As the implementation of LAPACK functions is based on BLAS, ScaLAPACK functions are largely based on parallel BLAS (PBLAS), which is closely integrated into the library [Bla+97; Int19]. Local operations can then be done using a BLAS library, which can be optimized for the specific platform. The actual communication between multiple processes is handled by Basic Linear Algebra Communication Subprograms (BLACS)[5], which optimizes communication tasks for linear algebra, such as sending and receiving matrices. BLACS is based on a communication protocol such as MPI, therefore it can be optimized and dependent on the specific platform it is used on.

[1]`https://github.com/SGpp/SGpp`
[2]`https://www.netlib.org/scalapack/`
[3]`https://www.netlib.org/lapack/`
[4]`https://www.netlib.org/blas/`
[5]`https://www.netlib.org/blacs/`

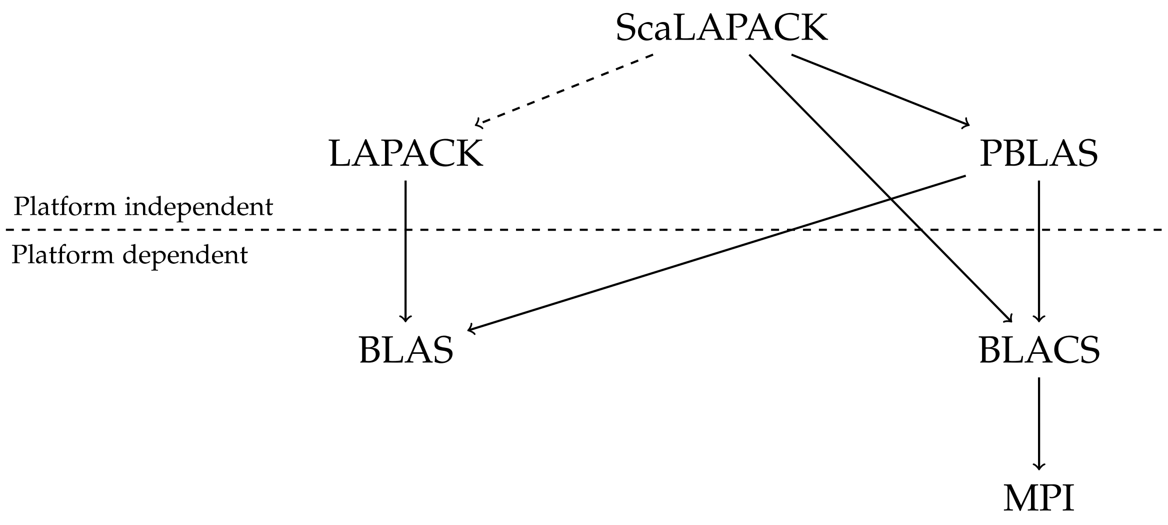


Figure 4.1.: Hierarchy of the ScaLAPACK library [Bla+97; Int19]. ScaLAPACK provides distributed versions of LAPACK routines, internally PBLAS is used to parallelize BLAS operations. Communication is based on BLACS, which in turn uses the MPI protocol. Portability is combined with efficiency by splitting the hierarchy in a platform dependent and independent part.

The structure of ScaLAPACK can also be represented as a hierarchy, as shown in Section 4.1. While the upper part of the hierarchy, ScaLAPACK and PBLAS, is platform independent, the lower part, which consists of BLAS, BLACS and a MPI library, is platform dependent. Therefore, the global, distributed interface is common among all platforms, while the underlying libraries that do most of the actual local processing, BLAS, BLACS and MPI, can be optimized for performance on a specific platform. This gives ScaLAPACK good portability while not sacrificing performance.

4.2. Matrix distribution

As noted above, calling Fortran routines from C++ code is not necessarily straightforward. The main reason for this is that Fortran uses column-major order for array access instead of row-major order like C++, which means that access to two-dimensional arrays is essentially transposed. This can become an problem for matrix operations, as a matrix created in C++ code appears as the transposed matrix to program parts written in Fortran. Furthermore, in Fortran indices start at 1 and in C++ at 0. To contain and resolve these problems in a small part of the code, two classes were introduced: *DataMatrixDistributed* and *DataVectorDistributed*.

These classes represent distributed matrices or vectors and replace the SG++ classes *DataMatrix* and *DataVector*, which represent local matrices or vectors. Additionally, the new classes provide a wrapper around common ScaLAPACK functions, which deals with the issues that arise from calling Fortran code from C++. This includes translating the indices and switching certain parameters to account for the transposed access to matrices. A performance loss by explicit transposition of matrices is also avoided. Consequently, the user of these classes does not have to worry about any of these problems.

Note that SG++ handles matrices and vectors in two separate classes. This enables interfaces to differentiate between matrices and vectors, which is useful as certain operations only support vectors or matrices. However in ScaLAPACK, distributed vectors are just handled as matrices with one column. Therefore, all matrix distribution methods are implemented in *DataMatrixDistributed*, whereas *DataVectorDistributed* is just composed of a matrix object with one column.

To distribute a matrix among multiple processes, communication functions are needed. This is provided by BLACS, which also manages the processes in a grid structure. The process grid is needed as ScaLAPACK uses a two-dimensional block cyclic matrix distribution scheme, as introduced in Section 3.4. To represent the process grid and provide access to common BLACS routines, the class *BlacsProcessGrid* was introduced.

On every process, an object of type *DataMatrixDistributed* stores a local part of the matrix according to its position in the process grid. Furthermore, the class provides methods to distribute or gather matrices, which send block after block of the matrix to the correct process using BLACS routines. As the access to the process grid and C++ arrays of local data is transposed in the Fortran routines, column and row indices are internally swapped in the *DataMatrixDistributed* and *DataVectorDistributed* class. This leads to the correct results without having to perform actual transpose operations, which would incur a performance loss.

A simplified class diagram of the components described in this section is shown in Figure 4.2.

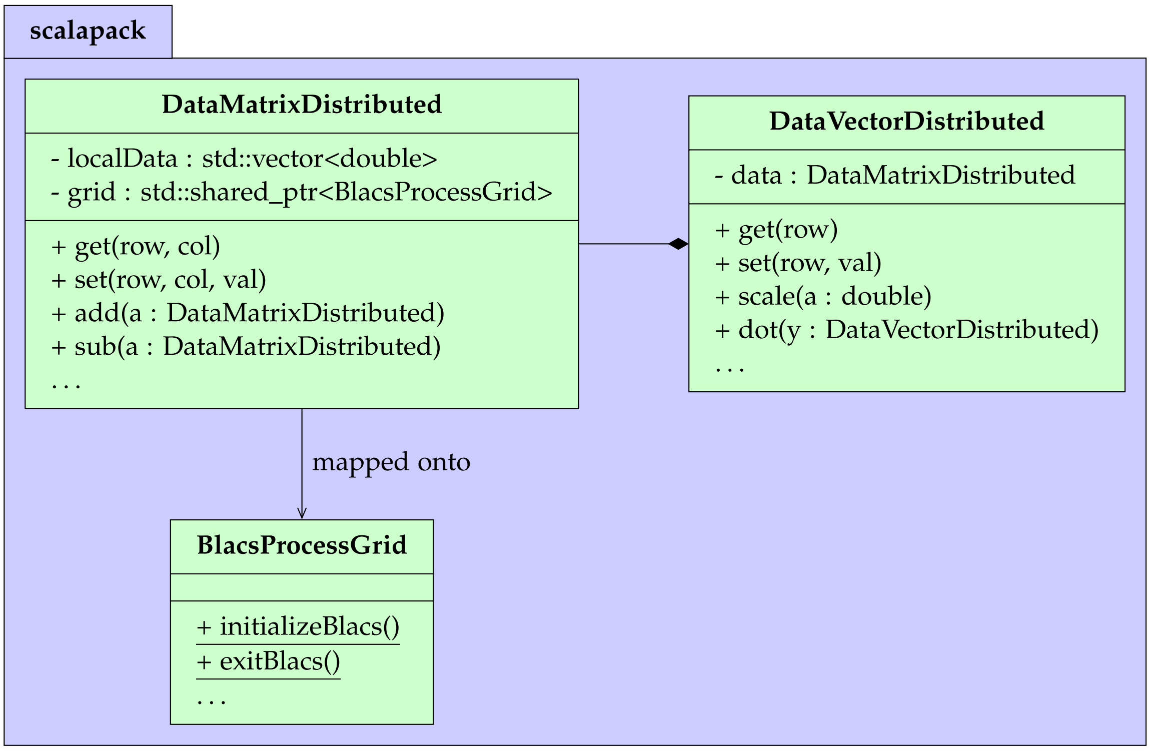

scalapack
DataMatrixDistributed
- localData : std::vector<double>
- grid : std::shared_ptr<BlacsProcessGrid>
+ get(row, col)
+ set(row, col, val)
+ add(a : DataMatrixDistributed)
+ sub(a : DataMatrixDistributed)
...
DataVectorDistributed
- data : DataMatrixDistributed
+ get(row)
+ set(row, val)
+ scale(a : double)
+ dot(y : DataVectorDistributed)
...
mapped onto
BlacsProcessGrid
+ initializeBlacs()
+ exitBlacs()
...

Figure 4.2.: New classes added for the Integration of ScaLAPACK into SG++. DataMatrixDistributed and DataVectorDistributed replace the classes DataMatrix and DataVector in the parallel version of the algorithm.

4.3. Integration of the Parallel Algorithm into the Datamining Pipeline

Next we look into the integration of the proposed parallel algorithm into the *datamining pipeline* of the SG++ library.

Datamining pipeline The datamining pipeline is a part of SG++ which allows users to easily execute different datamining tasks on sparse grids. Additionally, it provides a modular structure that can be extended easily. The SGDE based classifier was integrated into the datamining pipeline by [Fuc18]. A simplified class diagram of the pipeline is shown in Figure 4.3, however users only have to interact with classes derived from the *SparseGridMiner*. This class offers functionality to receive an extensive configuration through a JSON file and instantiate the proper modules required for the datamining task described in this configuration. Then, the learning process can be started and the results can be evaluated through this class.

Configuration As the parallel algorithm requires new configuration options, a *ParallelConfiguration* module was added. This allows to specify the shape of the process grid and the block size for the distribution scheme. When the SG++ library is compiled with the *USE_SCALAPACK* option and the parallel configuration options are found in the JSON file, the miner uses the parallel version of the algorithm.

Algorithm module Strictly speaking, the algorithm module is not part of the datamining pipeline itself, however it provides the actual computational operations. Both the computation of factorization of the system matrix as well as the solve operation for the linear system during the online phase are implemented in the algorithm module. Depending on the configuration of the pipeline, different factorization types can be used. As currently mostly the Cholesky and Orthogonal decompositions are used, the parallel algorithm was only implemented for these decomposition types.

DBMatOffline This class represents the factorization of the system matrix. It was extended in order to distribute the decomposed matrix among the processes and update this decomposition after refinement. The classes *DBMatOfflineChol* and *DBMatOffline-OrthoAdapt* inherit from it and were also adapted as needed for this purpose. Currently, all processes still have to hold the whole decomposition locally in addition to the distributed version, as this is needed for updates after refinements. This should be improved in future versions, as described in Chapter 6.

DBMatOnlineDE In order to handle the Online phase of the SGDE algorithm, this class is used. While the base class implements the general part of the computation of the density function, subclasses for the different decomposition types handle the solve step of the linear system. Parallel versions of the solve step using the distributed datastructures from the scalapack module were implemented in *DBMatOnlineDEOrthoAdapt* and *DBMatOnlineDEChol*. To use this interface, the method *computeDensityFunctionParallel* was added to *DBMatOnlineDE* in order to also makes use of the distributed datastructures. Additionally, functionality for distributed evaluation of the validation set on the model was added. As this and the computation of the vector $\boldsymbol{b}$ relies on evaluation of multiple datapoints on the sparse grid, the class *AlgorithmMultipleEvaluationDistributed* was added to provide the functionality of *AlgorithmMultipleEvaluation* in a parallel manner. As already noted in Algorithm 2, this requires a full local copy of $\boldsymbol{\alpha}$ on every process, as the recursive evaluation algorithm might require access to values which do not lie in the local part of $\boldsymbol{\alpha}$.

DBMatDecompMatrixSolver While the subclasses of DBMatOnlineDE update the decomposition in case of refinement and prepare the solve step, the actual solve operation for the linear system is implemented by subclasses of *DBMatDecompMatrixSolver*. Parallel solve operations were added to *DBMatDMSOrthoAdapt* and *DBMatDMSChol*. The new methods make use of ScaLAPACK and the distributed matrices and vectors introduced in the previous section to solve the linear system for $\boldsymbol{\alpha}$ in a parallel way.

ModelFittingDensityEstimationOnOffParallel Parallel density estimation was integrated into the pipeline by adding the class *ModelFittingDensityEstimationOnOffParallel* as a subclass of *ModelFittingDensityEstimation*. It provides the same functionality as *ModelFittingDensityEstimationOnOff*, however it adds a distributed version of the surplus vector $\boldsymbol{\alpha}$ and uses the parallel interface of *DBMatOnlineDE*. Furthermore, it initializes and synchronizes the local and distributed versions of $\boldsymbol{\alpha}$ and the matrix factorization when needed.

ModelFittingClassification The class *ModelFittingClassification* uses a density estimation model for each class to implement the SGDE based classification algorithm. The class was adapted in order to use parallel models when parallel configuration options are found. Furthermore, the *evaluate* method was changed in order to make use of the tasks managed by the process grid instead of using OpenMP in this case.

This class also handles the refinement of the models, however this was not yet changed, as refinement is still serial and is done on every process in order to avoid additional communication overhead.

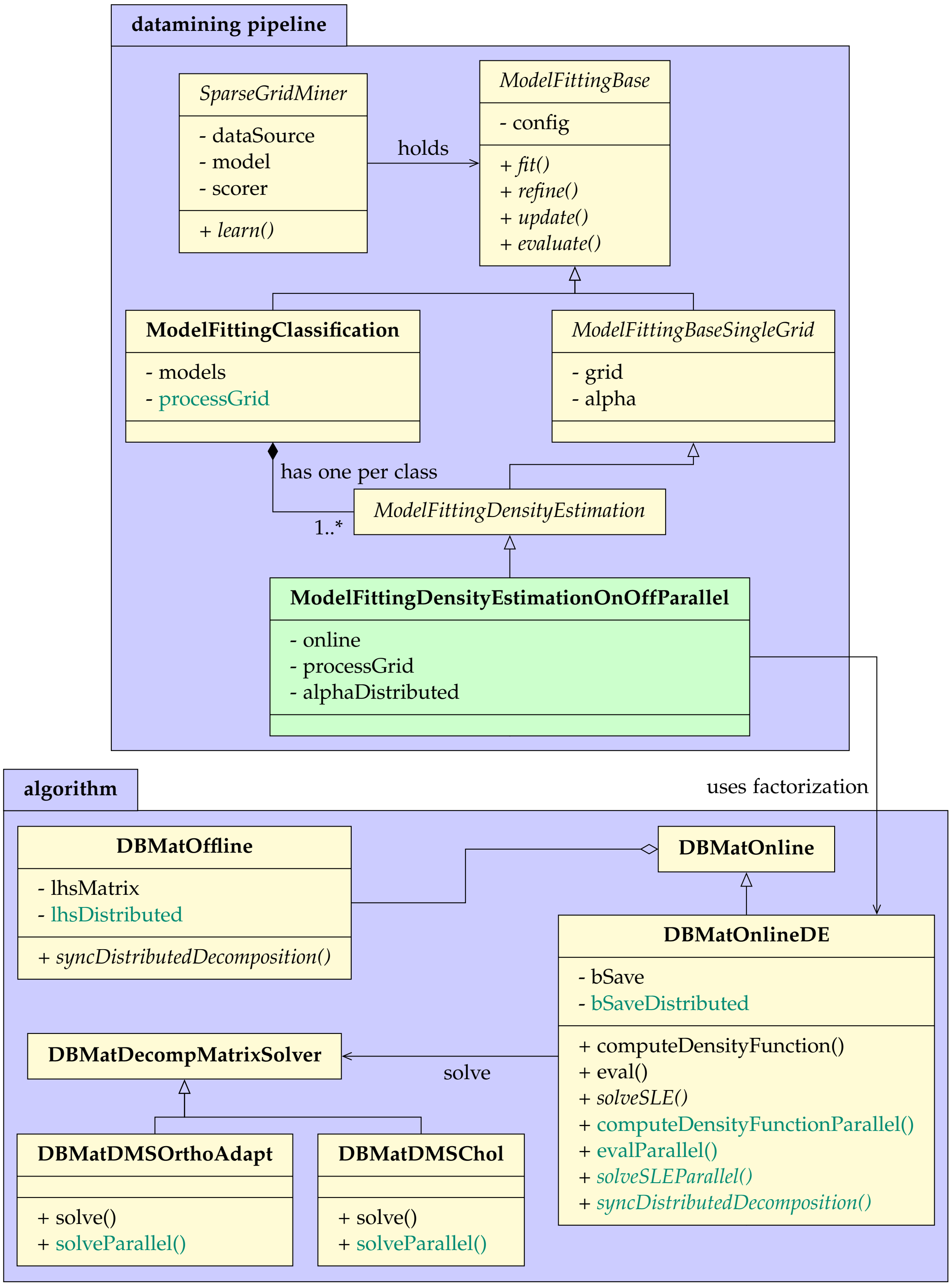


Figure 4.3.: Simplified class diagram of the datamining pipeline and algorithm module. Classes and methods that were added to implement the parallel algorithm are shown in green. Adapted from [Fuc18].

5. Evaluation

In this chapter, the parallel algorithm is tested and its performance is evaluated. Furthermore, general laws about parallel algorithms are presented. First, we consider the configuration of the algorithm and discuss bottlenecks, then we examine strong and weak scaling. Note that all performance evaluations presented here refer to the Online phase only, the system matrix decompositions were pre-computed and stored beforehand.

5.1. Analysis of the proposed algorithm

To begin with, the general behavior of the algorithm is analyzed. This means checking its correctness compared to the previous version, identifying good parameters for the configuration, and evaluating the MPI communication behavior. Note that all tests in this chapter were performed with *OMP_NUM_THREADS* set to 0 in order to avoid any interference of OpenMP and MPI tasks.

Correctness Given a correct implementation, the proposed algorithm, which is shown in Algorithm 2, should lead to the same results as the previous implementation. The mathematical operations are equal and there is no dynamic scheduling of the algorithm itself, so all operations should be deterministic. To check for implementation errors, unit tests were developed, especially the distributed datastructures were checked for correctness this way. Furthermore, tests on multiple datasets with different configurations were performed. For each run, the previous implementation and the new algorithm were compared using the same configuration. The results were always equivalent, the only differences observed were very small and insignificant rounding errors. Consequently, we can reasonably expect that the implementation is correct and the proposed algorithm returns the essentially same results as the previous version.

Configuration tuning The implementation of the parallel algorithm offers configuration options for the block size and the shape of the process grid. To explore the influence of different values for these options, some test runs were conducted. A configuration with 4 MPI tasks was chosen for this comparison.

First, different shapes of the process grid were tested. Assuming there are p^2 processes available, one possible choice would be a square $p \times p$ grid of processes, as this ensures an uniform distribution of matrices. However, vectors are only distributed onto one column of the process grid, which suggest that for matrix vector BLAS operations a process grid of shape $p^2 \times 1$ might be better suited. A $1 \times p^2$ grid would not be a good choice, as vectors are only mapped to a single process in this case. As the SGDE algorithm uses multiple vector operations, it is expected that a $p^2 \times 1$ grid delivers the best performance.

To confirm this expectation, multiple tests with an increasing numbers of processors from 1 up to 36 and a fixed problem size were performed. As expected, the configuration with a column-like process grid of $p^2 \times 1$ processors had the lowest runtime. The results were very similar for both Orthogonal and Cholesky decomposition for all configurations. When a square grid configuration of $p \times p$ processes was chosen, the performance was similar to a column shaped grid with $p \times 1$ processes, which is likely the result of vector operations only using the p processes of the first column. Row shaped process grids with $1 \times p^2$ processes show almost no performance improvement with increased number of processes, which is also as expected. In conclusion, the column-shaped process grid shape was found to be the best configuration for our algorithm by far, so all further tests were performed using this configuration.

Second, the influence of the block size for matrix distribution was explored. The block size should not be too big, as this could negatively impact load balancing and lead to very large MPI packets. It should also not be too small, as this could negatively affect efficiency of distributed operations and lead to a flood of small MPI packets. As [Bla+97] suggests to start with a block size of 64×64, block sizes of 32×32, 64×64 and 128×128 were compared.

The configuration with blocks of size 64×64 performed best, however the difference to the smaller block size of 32×32 was very small. Only the configuration with the larger block size of 128×128 performed notably worse. This indicates that overall, the exact block size does not seem to be very important, as long as it is not too large.

MPI Communication An important aspect of the behavior of the implementation is the MPI Communication overhead. It is desirable that the time spent on communicating and waiting on other tasks is as low as possible. Using the Intel Application Performance Snapshot tools, time spent in MPI routines was measured for configurations with 4 tasks and 32 tasks using the same global problem size. The absolute time spent on MPI communication was only marginally longer for the configuration 32 processes, which means there is almost no added MPI overhead when increasing the number of tasks. However, a sufficient problem size per node is necessary, as otherwise the fixed time needed for communication can make up a large part of total runtime, as this decreases with added tasks.

5.2. Scalability

To test how the parallel algorithm performs when the number of processes is increased, scalability testing has to be performed. Generally, we can differentiate between *strong scaling* and *weak scaling*. The former means that the *global* problem size stays constant, for the later the *local* problem size is fixed. Both types of scalability are explored in the following sections.

5.2.1. Strong Scaling

In the case of strong scaling we are interested in decreasing the runtime of the algorithm by keeping a constant global problem size and increasing the number of MPI tasks, which means there are more processes available.

Expectation As the runtime is expected to decrease, we are interested in the *speedup* $S = \frac{T_1}{T_p}$, where T_1 is the runtime of the serial program and T_p is the runtime of the program with p processes [Pac11]. In an ideal case, the speedup during the strong scaling test would be linear in the number of processes, so $S = p$. Most programs are not completely parallelized, but contain serial parts, therefore a linear speedup is hard to achieve. This is described by Amdahls law [Amd67], which can be formulated as

$$\frac{1}{r_s + \frac{r_p}{p}}. \tag{5.1}$$

With r_s and r_p being serial and parallel fraction of the program, so $r_s + r_p = 1$, the equation describes the effect that a large speedup can only be excepted if the serial overhead is quite small.

As the refinement of the grid in our algorithm is not yet parallelized, the expected speedup for configurations with extensive refinement is limited. Without refinement, there is only a smaller overhead, which consists mostly of setup operations, handling of the dataset and communication overhead for distributed operations. Given a sufficient problem size, the program should achieve a large speedup in this case.

To relate the actual speedup to the ideal performance, the *efficiency* $E = \frac{S}{T_p} = \frac{T_1}{p \cdot T_p}$ can be used, which can also be represented as a percentage.

Test setup The tests were run on the Linux Cluster *CoolMUC-2* provided by Leibniz-Rechenzentrum[1]. To keep the problem size and runtime for all tests reasonable, up to 32 MPI tasks on 4 nodes were used, the tasks were evenly distributed onto the nodes

[1]`https://doku.lrz.de/display/PUBLIC/CoolMUC-2`

for all tests. In the following tests, each configuration was run 10 times. The number of OpenMP threads was set to one to prevent interference with MPI tasks. First, the previous implementation was timed to set a reference, then the runs were conducted with the number of MPI tasks p equal to the powers of two from 1 up to 32. For the scaling tests, a dataset derived from Data Release 10 (DR10) of the Sloan Digital Sky Survey[2] project was used [Ahn+14]. The training set is four-dimensional and includes over 640 000 samples of two classes.

The strong scaling test was performed with and without refinement, furthermore Orthogonal and Cholesky decompositions were used. This lead to 4 different test configurations. In all configurations, training was done for one epoch with a batch size of 50 000 and $\lambda = 10^{-5}$. The block size of the two-dimensional block cyclic distribution was set to 64×64 and the process grid was shaped into a row in order to perform optimal results. For the versions with refinement, an error based refinement strategy with the *zerocrossing* refinement functor was used, per refinement 10 points were updated and there was a maximum of 10 refinements.

Results The results of the version without refinement can be seen in Figure 5.1, for the test with refinement in Figure 5.2. As expected, the scaling without refinement is significantly better, which follows Amdahls law. Furthermore, neither version has ideal linear scaling, as there is a some communication overhead for MPI and some parts of the code, such as initialization, setup of the grid, loading the data and decomposition is still serial. This is exaggerated for a larger number of processors, as the problem size per node gets smaller, which means the fraction of runtime needed for the overhead increases. This can also be measured using the efficiency, which drops from about 90% for 2 processes to just over 20% for 32 processes. Because of this, the runtime for 32 tasks stagnates and does not significantly decrease anymore. For a larger initial problem size, this problem could be overcome.

[2] *Official SDSS-III Acknowledgement*
Funding for SDSS-III has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Science Foundation, and the U.S. Department of Energy Office of Science. The SDSS-III web site is http://www.sdss3.org/.
SDSS-III is managed by the Astrophysical Research Consortium for the Participating Institutions of the SDSS-III Collaboration including the University of Arizona, the Brazilian Participation Group, Brookhaven National Laboratory, Carnegie Mellon University, University of Florida, the French Participation Group, the German Participation Group, Harvard University, the Instituto de Astrofisica de Canarias, the Michigan State/Notre Dame/JINA Participation Group, Johns Hopkins University, Lawrence Berkeley National Laboratory, Max Planck Institute for Astrophysics, Max Planck Institute for Extraterrestrial Physics, New Mexico State University, New York University, Ohio State University, Pennsylvania State University, University of Portsmouth, Princeton University, the Spanish Participation Group, University of Tokyo, University of Utah, Vanderbilt University, University of Virginia, University of Washington, and Yale University.

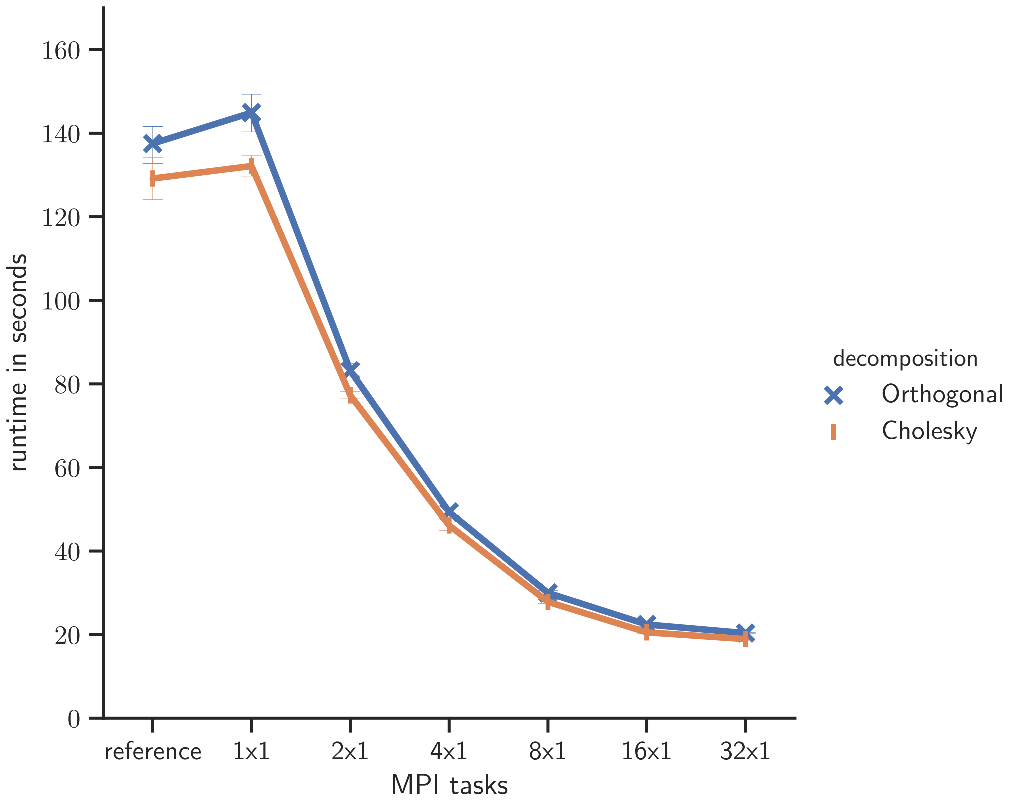


Figure 5.1.: Results of the strong scaling test on the dr10 dataset without refinement. Very similar performance for both Orthogonal and Cholesky decompositions is observed. Scaling efficiency starts out good and stagnates compared to ideal performance as the problem size per node gets smaller.

When looking at the version with refinement enabled, this effect is even more pronounced, as the serial fraction of the runtime is already very large initially. Note that there is an added communication overhead for the redistribution of the updated decomposition after refinement. As the performance gain in the parallel part stagnates, this overhead causes the overall runtime to even increase again. Furthermore, we can see that with refinement the version with Orthogonal decomposition is significantly slower than with the Cholesky decomposition. This is caused by the fact that more refinements are performed in the Orthogonal version, as the decision to refine is based on a heuristic.

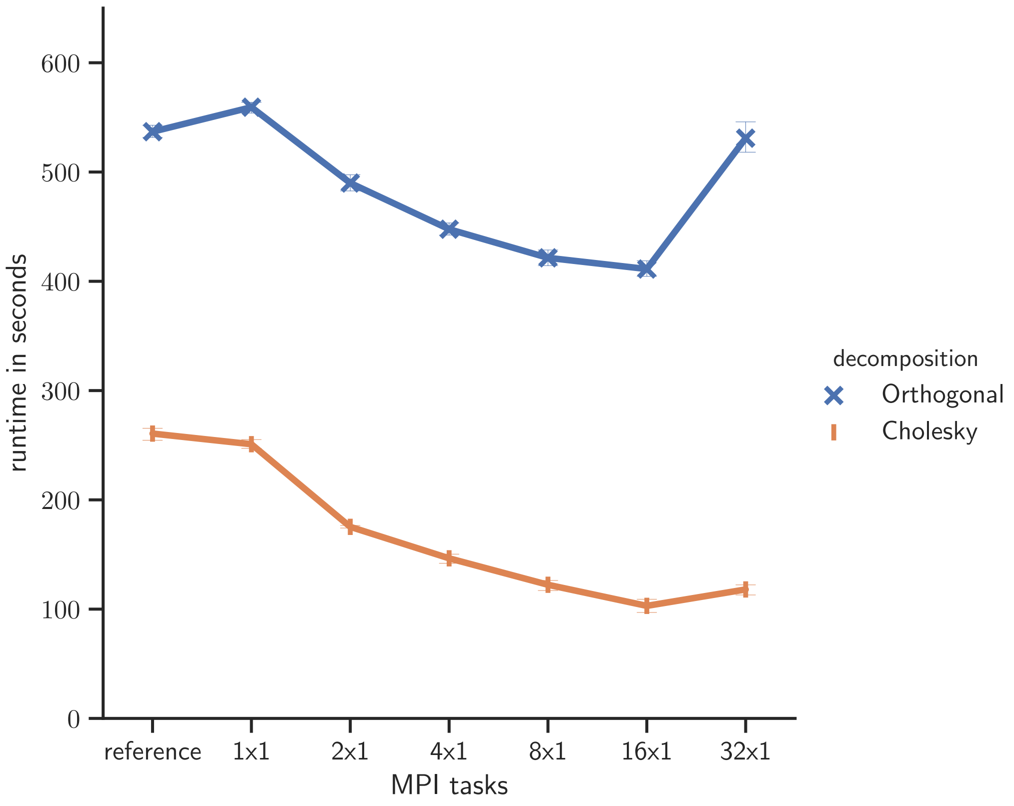


Figure 5.2.: Results of the strong scaling test on the dr10 dataset with refinement enabled. As refinement acts as a bottleneck, scaling efficiency is not very good. When the Orthogonal decomposition is used, the runtime for refinement is significantly higher than with Cholesky decomposition. For a high number of parallel tasks, the refinement step takes most of the runtime, so the overhead of further parallelization outweights any further gains.

Conclusion We can conclude that the algorithm scales as expected, however there is room for improvement as refinement leads to a large overhead. As a solution to this, the refinement part of the algorithm has to be parallelized as well, this is further described in Chapter 6.

5.2.2. Weak Scaling

In weak scaling, the goal is to test the performance of the program with increasing numbers of MPI processes while keeping the problem size per process fixed [Pac11]. The Online phase of the SGDE algorithm with a total of M datapoints, a batch size of m and a system matrix of size N has a theoretically runtime complexity of $\mathcal{O}(\frac{M}{m}(m + N^2)) = \mathcal{O}(\frac{m}{b}N^2)$. Therefore, to keep the problem size fixed per process we have to increase both M and m by the same factor.

Expectation As the problem size is constant per process, ideally the runtime should stay constant as well, regardless of the number of processes and global problem size. However in reality, this goal can often not be met entirely, as the communication overhead increases with number of nodes. Furthermore, for our implementation, memory consumption rises per node when the global problem size is increased, this might lead to a longer time needed for memory allocation or out of memory errors when the problem size grows too large.

Test setup For the weak scaling tests, the DR10 dataset was used again. To achieve the correct values for the total number of datapoints and batch size, only part of the dataset was used for each test. A validation portion of 0.1 was used for all runs, so the global size of the validation set grows, while the size per process also stays constant. The following table shows the global problem size for the weak scaling test from 1 up to 32 processes:

Number of processes	1	2	4	8	16	32
M	18000	36000	72000	144000	288000	576000
m	1800	3600	7200	14400	28800	57600
$\lvert\mathcal{S}_{train}\rvert + \lvert\mathcal{S}_{val}\rvert$	20000	40000	80000	160000	320000	640000

Weak scaling tests were conducted using Orthogonal and Cholesky decomposition, with and without refinement. The remaining configuration options were the same as for the strong scaling test. For each configuration, 10 runs were conducted.

Results The average runtime of the weak scaling test without refinement can be seen in Figure 5.3, the results of the test with refinement are shown in Figure 5.4.

For the test without refinement, there is a slightly growing overhead when increasing the number of processes. However, the overhead grows much slower than the global problem size. Furthermore, there is a relatively high variance between the runs, which might be explained by the short overall runtime. When doubling the number of tasks, there are also some combinations where the runtime decreases, especially for the

Orthogonal decomposition. This could be caused by external influences on the runtime or increased communication efficiency for certain numbers of tasks.

Note that with refinement, the weak scaling test is not quite well defined. As refinement increases the number of grid points, the local problem size grows, which is expected to increase the runtime. However, the decision to perform refinement is based on the error function, which usually decreases when a larger training set is used. Consequently the number of refinements can decrease as the number of tasks increases. This can also be observed in the sudden drops in runtime in the results of the weak scaling test with refinement. Note that more refinements are performed in the version with Orthogonal decomposition, which results in the slower runtime compared to the Cholesky decomposition. When ignoring the drops, the general trend is similar as without refinement. Comparing the overall runtime with the test without refinement, it becomes clear that refinement takes most of the runtime at this problem size. However, refinement also carries an additional communication overhead, as the system matrix factorization has to be redistributed after refinement. The increase of this overhead can be observed clearly observed when the number of processes exceeds 16.

Conclusion While there is a slightly growing overhead when increasing the number of processors, the weak scaling results were generally as expected. When compared to the growth of the global problem size, the increase of overhead is not significant. Furthermore, the local problem size in this test was quite small, as it was limited by the size of the dataset. With a larger problem size, the variance and fraction of overhead should further decrease.

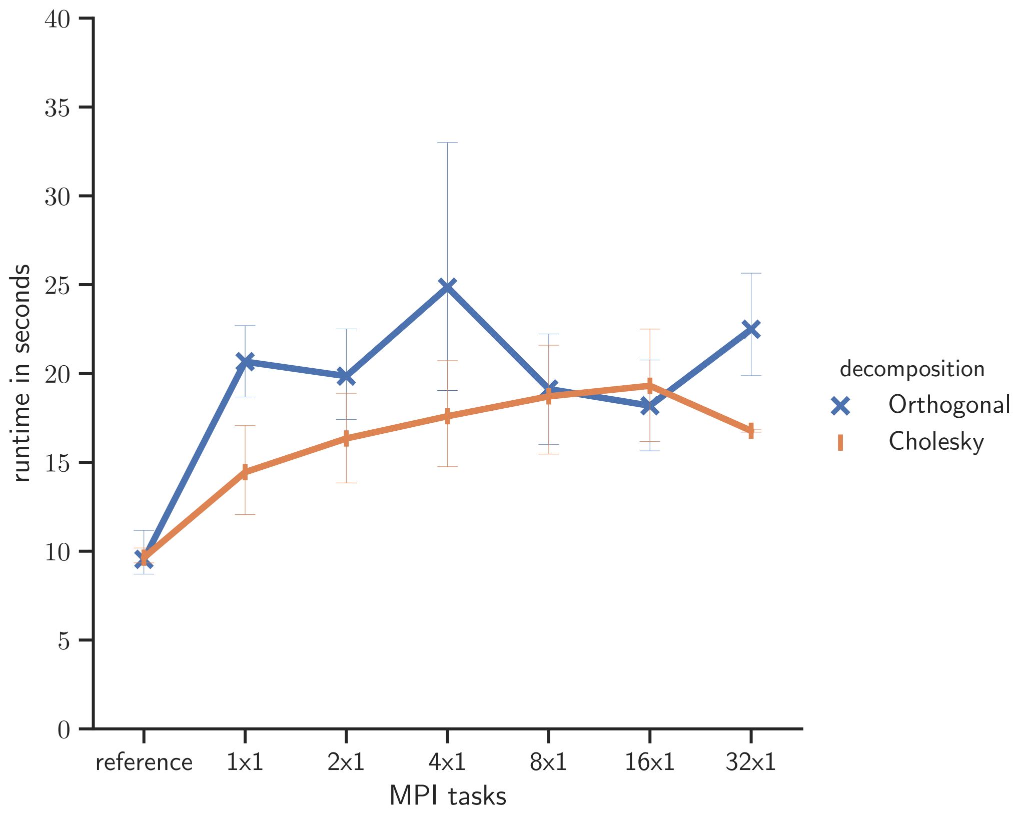


Figure 5.3.: Results of the weak scaling test on the DR10 dataset without refinement. The results show a higher variance between runs than for the strong scaling tests, which is likely caused by the relatively small problem size per node. Weak scaling efficiency shows only a small overhead when increasing the number of MPI tasks. The version with Cholesky decomposition performs slightly more consistent than Orthogonal decomposition, which is likely caused by a lower communication overhead, as fewer matrices have to be distributed.

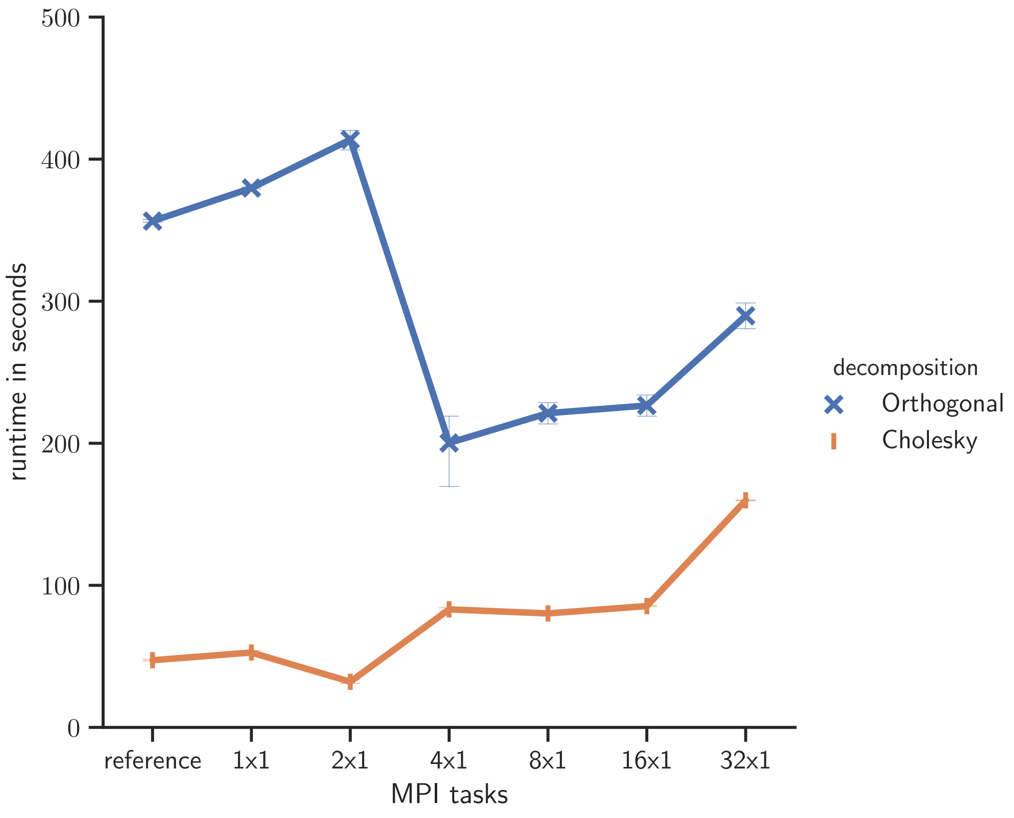


Figure 5.4.: Results of the weak scaling test on the DR10 dataset with refinement. For certain configurations, the runtime gets significantly lower. This is caused by increased classification accuracy, which is a result of the larger global problem size, leading to a lower number of overall refinements. However, with an increasing number of tasks, additional communication overhead for distributing the updated decomposition causes a drop in performance.

6. Future Work

As already noted at some points in the previous chapters, there are some areas where further improvements of the current algorithm and implementation are possible. This was outside the scope of this thesis, however it can be implemented in the future to enhance the performance of the algorithm presented here.

Parallelization of refinement As shown in the previous chapter, enabling refinement leads to a significant runtime overhead. This is mostly caused by the fact that refinement is currently done in serial. Consequently, a significant performance gain can be expected from parallelizing the refinement operation as well as the update of the system matrix decomposition that follows refinement. As the refinement function is currently called on all learners, integration of a parallel refinement algorithm should be possible. Depending on the factorization type used, implementing a parallel system matrix update should essentially involve translating conventional linear algebra operations to ScaLAPACK operations, which can be performed by the *DataMatrixDistributed* and *DataVectorDistributed* classes. However, these classes might need to be extended if further operations are needed.

Optimization of memory usage In addition to achieving a lower runtime, parallelization of the system matrix updates can also be used to lower memory usage. Currently, *DBMatOffline* class has to store both a local and a distributed version of the factorization. For the Orthogonal decomposition, there are even multiple matrices in the *DBMatOfflineOrthoAdapt* and *DBMatOnlineOrthoAdapt* classes that are stored in this redundant manner. When a local refinement and matrix factorization update is no longer needed, these classes can be refactored to eliminate the local storage during the Online phase and therefore reduce memory usage.

Vectorization of mass evaluation To create the vector $\boldsymbol{b}$ and for calculation of the error function, all datapoints of the training or validation set have to be evaluated on the sparse grid. While this is parallelized in the new algorithm, it currently uses a highly recursive evaluation algorithm. While this algorithm is optimal in theory, it has been shown that using vectorization of modern hardware to directly calculate the interpolation $\sum_i \alpha_i \varphi_i$ is faster [HP11]. Therefore, it should be explored whether it is possible to exceed the performance of the current implementation by parallelizing the vectorized version of the evaluation routine.

Parallelization of the Offline phase While not related to the performance of the Online phase, the runtime of the Offline phase can get unreasonable long for high dimensional problems. This could be solved by using ScaLAPACK to parallelize the calculation of the matrix decomposition.

7. Conclusion

As a result of this thesis, a model-parallelization approach was successfully implemented for the SGDE based classification algorithm in the SG++ library. This involved evaluation of different parallelization approaches, design of a distributed algorithm, analysis of the matrix distribution scheme and implementation, including integration into the datamining pipeline. Furthermore, the ScaLAPACK library was integrated into the SG++ toolbox and easy to use wrapper classes were created for the distributed data structures. This provides a good basis for futher parallelization projects. Some areas where this would be beneficial have been identified in the previous chapter.

The performance of the implementation was analyzed and evaluated with strong and weak scaling tests. Results were largely as expected and only degraded when insufficient problem sizes were reached. However, it was identified that refinement creates a bottleneck which severly limits speedup, consequently this area should have high priority for further parallelization.

Overall, an efficient parallel implementation of the classifier was combined with the highly configurable and easy to use datamining pipeline. It is expected that this enables efficient learning even on highly dimensional datasets with a large number of grid points. Therefore, the capabilities of the SGDE based classifier in the SG++ library were increased by this addition.

A. Appendix

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	149.81	136.61	90.53	53.79	29.55	21.80	20.18
2	141.32	132.60	83.26	47.95	29.83	22.57	20.24
3	140.53	152.44	81.04	49.33	29.47	21.92	20.40
4	140.11	137.27	81.76	50.19	30.59	21.74	20.18
5	129.97	151.27	82.53	48.77	29.69	23.83	20.03
6	128.32	137.84	81.38	48.82	30.12	21.84	20.48
7	124.24	150.92	83.53	48.03	29.72	23.15	20.98
8	140.08	150.53	83.13	48.96	29.32	22.50	20.11
9	140.48	150.76	81.61	49.29	29.39	21.65	20.68
10	140.40	148.99	82.63	48.18	32.13	23.15	20.38

Figure A.1.: Results of the strong scaling test without refinement and with Orthogonal decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	140.49	127.59	80.41	50.67	27.26	20.16	18.69
2	136.33	127.23	76.36	44.68	28.47	20.44	18.59
3	136.50	136.08	76.41	45.26	27.28	20.27	18.78
4	136.37	128.05	76.06	46.43	27.33	21.77	18.72
5	125.19	135.46	77.07	44.69	27.89	20.45	18.68
6	120.74	134.10	77.68	44.63	27.62	20.35	19.63
7	121.32	135.73	77.29	45.13	29.55	20.88	19.17
8	136.64	135.86	77.81	49.94	27.23	20.34	18.57
9	119.03	126.98	77.34	45.39	28.10	20.30	19.81
10	118.99	134.38	76.06	44.43	27.59	20.42	18.62

Figure A.2.: Results of the strong scaling test without refinement and with Cholesky decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	554.24	553.70	505.21	445.99	411.80	401.74	533.63
2	535.97	564.79	500.98	463.15	413.81	401.38	490.41
3	531.31	561.85	473.52	443.58	410.90	423.13	521.82
4	531.23	559.26	503.16	445.79	434.67	401.75	526.97
5	551.22	547.41	481.35	456.54	411.81	404.37	521.19
6	546.08	557.40	505.96	448.59	435.20	427.24	519.49
7	530.18	568.28	490.24	461.26	412.30	402.58	527.19
8	528.97	561.59	476.43	437.89	438.85	402.28	564.83
9	528.54	572.29	477.38	437.63	433.87	421.38	566.76
10	531.22	546.59	485.18	436.27	411.61	426.64	537.55

Figure A.3.: Results of the strong scaling test with refinement and with Orthogonal decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	264.94	258.18	180.42	135.05	124.02	95.86	120.47
2	274.25	264.18	174.33	151.50	125.39	94.40	100.69
3	262.49	246.51	176.73	135.61	128.72	94.69	122.60
4	239.13	243.40	174.83	149.34	125.72	115.50	103.36
5	259.35	253.03	173.57	151.95	108.24	115.10	121.51
6	259.30	253.56	173.90	150.90	127.28	94.62	123.20
7	264.80	243.39	173.75	151.42	125.63	94.99	121.74
8	255.49	246.74	174.84	149.04	127.01	95.00	122.16
9	266.89	249.57	174.15	150.12	123.69	114.89	121.14
10	259.86	250.76	176.51	140.12	107.34	114.97	123.24

Figure A.4.: Results of the strong scaling test with refinement and with Cholesky decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	16.85	22.73	22.35	56.92	14.95	14.66	18.17
2	8.69	17.55	25.47	30.12	24.02	24.87	29.41
3	8.70	22.56	16.14	25.53	14.02	14.61	18.36
4	8.70	17.66	17.81	15.49	25.19	14.50	29.03
5	8.70	22.63	16.53	26.04	13.68	14.91	18.37
6	8.74	17.86	25.49	15.64	23.95	24.69	23.19
7	8.76	22.95	17.41	26.18	13.95	14.61	18.33
8	8.69	17.59	25.45	15.54	23.89	24.63	23.33
9	8.79	17.57	15.61	20.15	13.88	14.77	18.34
10	8.81	27.57	16.27	16.91	23.74	19.64	28.38

Figure A.5.: Results of the weak scaling test without refinement and with Orthogonal decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	12.07	11.64	13.69	16.26	16.67	18.14	16.98
2	9.44	11.69	21.00	17.29	11.61	13.01	16.73
3	9.38	11.71	10.87	12.33	11.63	23.44	16.69
4	9.33	17.00	20.77	12.45	23.03	29.43	16.55
5	9.29	12.41	10.75	23.07	21.67	23.09	16.87
6	9.39	12.00	20.99	22.73	23.58	23.19	16.67
7	9.39	22.40	15.78	13.04	22.09	18.06	16.75
8	9.31	11.93	11.07	12.50	21.75	12.99	16.84
9	9.46	11.63	20.81	22.54	23.48	18.52	16.71
10	9.32	22.07	17.72	23.76	11.63	13.24	16.99

Figure A.6.: Results of the weak scaling test without refinement and with Cholesky decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	360.31	380.75	427.99	220.05	232.68	236.56	262.26
2	356.05	379.54	419.18	207.52	208.79	211.42	291.03
3	357.70	378.19	418.72	211.45	211.85	237.27	281.97
4	355.48	381.90	395.08	207.34	233.03	211.48	309.42
5	355.63	379.84	420.65	206.39	235.02	211.30	281.19
6	355.33	379.57	419.19	240.32	232.13	236.11	314.41
7	355.22	379.12	403.96	74.37	207.40	212.73	306.44
8	356.13	378.83	418.43	213.59	206.87	236.87	288.19
9	355.74	379.02	394.52	207.01	234.49	235.99	283.96
10	356.17	380.03	419.37	214.29	209.80	236.21	278.58

Figure A.7.: Results of the weak scaling test with refinement and with Orthogonal decomposition. Runtime for the 10 runs is given in seconds.

	MPI tasks						
run #	reference	1×1	2×1	4×1	8×1	16×1	32×1
1	50.45	52.11	33.74	87.28	79.17	85.15	160.15
2	46.90	51.42	30.67	81.76	79.71	84.89	160.06
3	46.60	54.26	30.37	84.60	79.13	84.99	159.53
4	47.07	51.77	30.40	82.03	79.06	86.23	159.92
5	46.94	58.26	33.19	82.08	82.71	86.76	159.66
6	46.64	52.12	32.22	84.23	78.90	85.37	160.19
7	46.88	51.44	30.35	81.84	82.61	85.28	159.55
8	46.69	51.31	32.28	82.37	79.78	85.73	159.74
9	46.89	51.90	37.29	82.71	78.88	84.84	160.49
10	47.63	52.56	30.76	81.87	82.27	84.90	159.09

Figure A.8.: Results of the weak scaling test with refinement and with Cholesky decomposition. Runtime for the 10 runs is given in seconds.

Bibliography

[Ahn+14] C. P. Ahn, R. Alexandroff, C. A. Prieto, F. Anders, S. F. Anderson, T. Anderton, B. H. Andrews, É. Aubourg, S. Bailey, F. A. Bastien, et al. "The tenth data release of the Sloan Digital Sky Survey: first spectroscopic data from the SDSS-III Apache Point Observatory galactic evolution experiment." In: *The Astrophysical Journal Supplement Series* 211.2 (2014), p. 17.

[Amd67] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[BG04] H.-J. Bungartz and M. Griebel. "Sparse grids." In: *Acta numerica* 13 (2004), pp. 147–269.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[Bla+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-397-8 (paperback).

[Bod17] V. Bode. "Parallelization of a Sparse Grids Batch Classifier." Bachelor's thesis. Technical University of Munich, 2017.

[Bos17] D. Boschko. "Orthogonal Matrix Decomposition for Adaptive Sparse Grid Density Estimation Methods." Bachelor's thesis. Technical University of Munich, 2017.

[Fuc18] D. Fuchsgruber. "Integration of SGDE-based Classification into the SG++ Datamining Pipeline." Bachelor's thesis. Technical University of Munich, 2018.

[HP11] A. Heinecke and D. Pflüger. "Multi- and Many-Core Data Mining with Adaptive Sparse Grids." In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. New York, USA: ACM, May 2011, 29:1–29:10. ISBN: 9781450306980.

[Int19] Intel(R). *Intel Math Kernel Library Developer Reference*. 2019.

[Kum02] V. Kumar. *Introduction to Parallel Computing*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201648652.

[Let17] M. Lettrich. "Iterative Incomplete Cholesky Decomposition for Datamining using Sparse Grids." Studienarbeit/SEP/IDP. Technical University of Munich, 2017.

[Mur12] K. P. Murphy. *Machine learning: a probabilistic perspective*. en. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012. ISBN: 978-0-262-01802-9.

[Pac11] P. Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[Peh13] B. Peherstorfer. "Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques." Dissertation. München: Technische Universität München, 2013.

[Pfl10] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. München: Verlag Dr. Hut, Aug. 2010. ISBN: 9783868535556.

[PPB10] D. Pflüger, B. Peherstorfer, and H.-J. Bungartz. "Spatially adaptive sparse grids for high-dimensional data-driven problems." In: *Journal of Complexity* 26.5 (Oct. 2010). published online April 2010, pp. 508–522. ISSN: 0885-064X.

[Sie16] A. Sieler. "Refinement and Coarsening of Online-Offline Data Mining Methods with Sparse Grids." Bachelor's thesis. Technical University of Munich, 2016.