Parallelization of a Sparse Grids Batch Classifier

Vincent Bode
I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

Munich, 14.09.2017

Vincent Bode
Abstract

The aim of this bachelor’s thesis is to evaluate, implement, and analyze the parallelization of a sparse grids batch classifier. Several potential methods of distributing the workload were identified and evaluated for their feasibility, including class-based, spatial, and batch-based methods. From this analysis, a parallelization scheme that distributes work by batches of training data was chosen due to the flexibility it offers for less than ideal datasets, even though it involves more overhead. A design for the overall structure of the program, that makes use of a master/worker role distribution, and a communication plan were created and subsequently implemented. The semantics of the implementation were shown using software architectural techniques and several challenges were presented. Next, a guide for tuning the performance of the new learner was introduced, showing which variables need to be adjusted to decrease the time to solution on multi-node systems. Using strong and weak scaling tests, the implementation was then tested for consistency and efficiency. Results that scale almost linearly, or in one case super-linearly, were observed when comparing the old sequential to the new parallel implementation. However, it was also shown that certain scenarios could be constructed where the new learner cannot harness the full potential on all nodes due to the sequential nature of the refinement process, which unfortunately was out of the scope of this thesis. This, and several other open points for future improvements were additionally examined to optimize the learner for even more use cases.
Zusammenfassung

Contents

Abstract iii

1. Introduction and Motivation 1

2. Classification 3
   2.1. Classification and Data 3
   2.2. Dimensionality 3
   2.3. Dimension reduction 5
   2.4. Grid Based Techniques 6
      2.4.1. Dimensional Adaptivity 6
      2.4.2. Spatial Adaptivity 6
      2.4.3. Overfitting 7
      2.4.4. More curse of dimensionality 7

3. Sparse Grids 9
   3.1. Introducing SG++ 9
   3.2. Basis Functions 9
   3.3. The Subspace Scheme 10
   3.4. Function interpolation 11
   3.5. Density Estimation 12
   3.6. Spatial Adaptivity 13
   3.7. The Cholesky Decomposition 13

4. Parallelization 15
   4.1. Existing solutions 15
   4.2. Parallelization approaches for the new learner 15
      4.2.1. Classes 16
      4.2.2. Spatial 16
      4.2.3. Batches 16
      4.2.4. Evaluation 18
   4.3. State of the Learner 18
   4.4. Distribution of Roles 18
   4.5. Refinement 20
   4.6. MPI-Communication 20

5. The Implementation 23
   5.1. Implementation Details 23
Contents

5.2. Implementation Challenges ........................................... 23
  5.2.1. Communication .............................................. 29

6. Communication-Plan .................................................. 31
  6.1. MPI Request Pool ............................................... 31
  6.2. The Scheduler ................................................... 35
  6.3. Alpha Merge Request ........................................... 36
  6.4. The Grid Update Request ....................................... 37
  6.5. System Matrix Update .......................................... 38

7. Analysis .............................................................. 39
  7.1. Classification Accuracy ......................................... 39
  7.2. Tuning ............................................................ 40
    7.2.1. Idle Times ................................................ 41
    7.2.2. Communication ............................................ 46
  7.3. Calibration of refinement to batches ......................... 47
  7.4. Scalability Testing ............................................. 48
    7.4.1. Conducting Tests ......................................... 48
    7.4.2. Ideal Performance ....................................... 48
    7.4.3. Simple strong scaling test ............................... 49
    7.4.4. Complex strong scaling test ............................ 50
    7.4.5. Weak scaling test ....................................... 52

8. Further improvements ................................................. 59
  8.1. Streaming data sets ............................................ 59
  8.2. Improve System Matrix Updates ................................ 59
  8.3. Parallel-Refinement ........................................... 60

9. Conclusion ............................................................ 61

Appendices ............................................................... 63

A. Experiment Results ................................................ 65

Bibliography ............................................................ 69
1. Introduction and Motivation

Over the last decades, knowledge acquisition has grown ever more important in the quest for rational and data-based decision making. The goal of making as many informed choices as possible, however, requires the information used to be readily available. This is where knowledge discovery through data mining is becoming increasingly important.

Already, from 2011 to 2016, the market revenue of big data has more than tripled to over 30 billion US dollars. A Statista forecast suggests that until 2026, this market is expected to triple in revenue once again from its standing in 2016, as storing and processing massive datasets becomes much cheaper [1]. However, the growing ability to handle large sets of data is not only driven by the hardware’s storage and processing capacity. It is also facilitated by the adaptation of software to use the available resources in a more efficient manner.

This need to process ever growing datasets requires data driven applications to utilize the available processing power as effectively as possible when striving to solve these problems accurately. Currently, the growth in processing power for high performance computing applications is provided for not by increasing a processors speed, but by adding processing units to increase the throughput in parallel. But doing so does not inherently help solve problems faster, unless the solver is taught how to use these parallel resources to decrease the time required for the problem’s solution.

One of these problems used for knowledge acquisition is classification, the ability to predict a discrete output variable from one or more input variables. A library that solves classification tasks is SG++, which uses sparse grids to solve several high performance computing tasks. Such applications may include clustering, differential equations, or said classification. In classification, training requires large sample data sets to allow the algorithm to adapt to the problem. It is typically a very computationally expensive task to achieve high accuracy on non-trivial data sets.

Currently, the learning implementation of the classifier implemented in SG++ is only sequential, limiting its potential application for large data sets. The aim of this thesis is to analyze, through theory presented in Sections 2 and 3, and parallelize the current implementation, by designing and implementing the new algorithm in Sections 4 through 6. Furthermore, Section 7 analyzes some efficiency metrics of the new implementation and Section 8 illustrates potential adjustments for future improvement.
2. Classification

2.1. Classification and Data

Classification is the algorithmic task of predicting a discrete output variable, the class, using one or more input variables. For a classification task, there is a correlation between the input variables and the output variables that allows prediction to occur. Let $\vec{x}$ be the vector of input variables, with its elements normalized into the range $[0, 1]^d$. This vector is $d$ elements long, equal to the dimensionality. The discrete output variable $k \in K$ is the class index of a data point in the set of all classes, the observed or predicted result among $|K|$ different possible classes. A multiset $S$ of data points is referred to as a data set,

$$S = \{(\vec{x}, k) | \vec{x} \in [0, 1]^d, k \in |K|\}. \quad (2.1)$$

Several examples of such data sets that will be used to perform experiments can be seen in Figures 2.1 and 2.2. A classification problem has a training set $S_{\text{train}}$, used for a solver to learn the correlation between the input variables $x$ and the output class $k$. In order to test how effectively the correlation has been learned, a test set $S_{\text{test}}$ can be used. Here, the solver is only given the input variables $\vec{x}$ from the test set $S_{\text{test}}$ to create a prediction $k_{\text{predicted}}$. This is done for every data point in the test set $S_{\text{test}}$ to generate the predicted test results $S_{\text{predicted}}$:

$$S_{\text{predicted}} = \{(\vec{x}, k_{\text{predicted}}) | \exists k_{\text{actual}} : (\vec{x}, k_{\text{actual}}) \in S_{\text{test}}, k_{\text{predicted}} \in K\}. \quad (2.2)$$

From this, the set of correct predictions can be extracted:

$$S_{\text{prediction\_correct}} = S_{\text{predicted}} \cap S_{\text{test}}. \quad (2.3)$$

The size of the set of correctly predicted data points is then used to calculate the accuracy for a given classifier and test set,

$$a = \frac{|S_{\text{prediction\_correct}}|}{|S_{\text{test}}|}. \quad (2.4)$$

Accuracy is one performance indicator of a classifier with a domain in $[0, 1]$.

2.2. Dimensionality

The requirements in data classification have increased with the rise of Data Mining as a means to acquire new information from data sets. As collecting relevant input for the
2. Classification

Figure 2.1: The 2 dimensional Ripley-Garcke dataset. The Ripley-Garcke dataset has only 1000 data points in the training set and a significant level of noise. It will be used to perform classification accuracy testing against the implementation of the original classifier as it emulates real-world dataset issues well. This dataset was original from the Ripley Repository [2] and modified by Jochen Garcke [3] for SG++.

classification task becomes cheaper (i.e. more sensors measuring different aspects of the environment), the classification algorithm needs to consider more complex data in order to complete its learning and testing phases. This is the so called *curse of dimensionality*. It stems from the interpretation that each added input variable \( \vec{x}_i \) for a data point \( s \in S \) increases the problem space \([0, 1]^d\) by one dimension. This is not dependent on whether the input has a small (binary) or huge domain, the increase in dimensionality is nevertheless constant.

Take a full grid as an example. If the number of samples \( p \) per dimension is kept constant, then the total number of grid points for grid \( G_{\text{full}} \) is equal to

\[
|G_{\text{full}}| = p^d.
\]  

(2.5)

Classification algorithms that are based on full grids struggle with this, as they generally increase in complexity linearly with the number of grid points and thus exponentially with respect to the number of dimensions \( d \).
2.3. Dimension reduction

One approach to solve high dimensional classification tasks is using dimension reduction. It relies on the assumption that problems formulated in a high dimensional space may also be solved in a space of much lower dimension. One technique in dimension reduction is use dimension elimination. To solve a problem this way, an algorithm would attempt to eliminate as many dimensions $d$ in $\vec{x}$ as possible, while attempting to prevent the variance in the output variable $k$ from decreasing significantly as inputs are eliminated.

This could be interpreted as examining the data set $\mathcal{S}$, weighing as many of the variables $\vec{x}_i$ as possible as irrelevant and learn from the few variables that remain, which are considered important. The interpreted importance of a variable $\vec{x}_i$ is the measure of how much a change in the input of a specific variable affects the observed class $k$, keeping all other input variables constant. For every dimension eliminated this way, the asymptotic complexity is reduced by $p$.

While dimension elimination can work really well for specific problems, where much of the data is less relevant, it struggles with other problems that cannot be reduced to a level necessary to solve them effectively. Note that other methods of dimension reduction exist that can exploit patterns in data to reduce problem dimensionality without requiring...
full elimination of dimensions.

2.4. Grid Based Techniques

So, instead of attempting to reduce the problem space, it can be desirable to solve the problem in the full problem space without reducing the dimensionality first; assume all the input can be relevant. To achieve this goal, grid based sampling methods will be examined. One possible approach previously mentioned is to use a full grid $G_{\text{full}}$ with $p$ equidistant grid points to sample a given problem space to learn the probability density functions. This has the advantage of a really simple implementation and works well enough for small problems in low dimensional spaces. However, with every new dimension in the problem, complexity quickly increases past the point of feasibility as full grids suffer the full curse of dimensionality.

2.4.1. Dimensional Adaptivity

To slightly improve this approach, it could be assumed that some dimensions of the problem will be more important to solving it than others, similar to the assumption of dimension reduction. The number of samples in each dimension $\vec{p}_i$ can be adjusted for dimension $i$ independent of the other dimensions. This density of grid points $\vec{p}_i$ that is specified in each dimension directly correlates to how important the dimension’s influence on the result is deemed. Thus, less grid points are needed to sample in the less important directions, saving time and memory. Now, the grid $G_{\text{full}}$ only has a size of

$$|G_{\text{full}}| = \prod_{i=0}^{p} \vec{p}_i. \tag{2.6}$$

Compared to dimension reduction, this method has the advantage of not completely eliminating the effect of less important dimensions.

2.4.2. Spatial Adaptivity

Continuing the line of thought used in the simplification above, one may realize that not only dimensions lack interesting properties, but often entire regions. Therefore, it is desirable for the algorithm to focus on regions that are deemed interesting and spend only little computation time and space on the regions that are not. This makes a spatially adaptive grid suited to a slightly different problem set than the dimensional adaptive grid. In classification, the regions of interest lie just at the boundary between regions that are classified into different classes. Here, it is desirable for the classifier to have a high resolution to accurately portray fine patterns in the data that can have a strong influence on the classification accuracy. Other regions, usually towards the edge of the problem space, require less resolution, as the classifier already has a high confidence in the classification result.
2.4. Grid Based Techniques

Figure 2.3.: Adaptive Refinement on Full Grids. First, a set of grid points needs to be found to refine (left). Next a full grid is placed inside this region (right), increasing the local density of grid points.

A solution to this problem using grids is called adaptive refinement. It works by first determining the interesting regions on the grid using a heuristic and then adding more grid points to those regions to increase the resolution. The introduction of this technique however brings two problems. First, it is not obvious how such a heuristic, needed detect these interesting regions for refinement, should work. Second, all the algorithms that work well on regular grids need to be adapted to work with these highly irregular grids so that the power of adaptive refinement can be leveraged.

2.4.3. Overfitting

It is with the spatially adaptive refinement that special care needs to be exercised to avoid overfitting. Overfitting occurs when the function learned by the grid resembles the training set too closely, replicating noise and jitter accurately instead of the probability density function, and thus does not perform well when faced with new datasets. A common remedy for this is to use a test set different from the training set and optimize the classification accuracy against both the tests.

2.4.4. More curse of dimensionality

Both dimensional adaptivity and spatially adaptive refinement have allowed a delay of when the curse of dimensionality starts to affect the ability to solve a problem in a realistic time frame. Both are suited well for specific problem sets. However, both these adaptive techniques are limited by the rigid structure of a full grid and cannot remove the problem that full grids eventually do suffer the curse of dimensionality. Instead, other grid based techniques may be examined. Sparse grids are one technique that allow the curse of dimensionality to be circumvented.
3. Sparse Grids

Regular full grids have a constant density $p$ in every part of the grid, resulting in a highly regular structure. Sparse grids, however, manage to avoid the full curse of dimensionality by allowing for different areas in a grid to have differing densities. Through this, a cost-benefit analysis can be made to determine the optimal densities of the sparse grid.

3.1. Introducing SG++

In this thesis, the usage of the SG++ library on parallel systems for classification purposes will be examined using density estimation and batch learning. SG++ is developed at the Chair of Scientific Computing in Computer Science at the Technical University of Munich and at the University of Stuttgart. It is an open source C++ implementation of sparse grids for various purposes. Among them are solvers for data mining problems such as classification. For a general reference to SG++, see [5]. SG++ has a current implementation of a batch learner to solve classification tasks sequentially. Credit for the original implementation goes to Peherstorfer [6] and Röhner [7], the subsequent integration into SG++ was performed by Maier [8]. The recent overhaul originates from Michael Lettrich in [9]. Over the course of the thesis, this batch learner will be decomposed and parallelized in order to minimize the time to solution. However, before the batch learner can be parallelized, some more theory needs to be covered.

3.2. Basis Functions

Sparse grids are structured in hierarchies to simplify working with regions where the density of grid points varies. To allow interpolation, a method to calculate function values of an arbitrary point in space on the grid is required. To achieve this goal, basis functions $\varphi_{l,i}(x)$ are used. A simple type of basis function to use is the linear hat function:

$$\varphi(x) = max(1 - |x|, 0). \quad (3.1)$$

These are translated horizontally so that they center around a grid point by an index $i$. Next, the $l$ index scales these functions horizontally, decreasing with an increase in depth of the hierarchy. With these inputs, the following family of functions is constructed:

$$\varphi_{l,i}(x) := \varphi(2^l x - i). \quad (3.2)$$
3. Sparse Grids

Figure 3.1.: A 1 dimensional level index vector. The grid points on the grid itself are addressed using an invisible tree like hierarchy that identifies their position uniquely. When using adaptive refinement techniques, this tree will no longer be balanced, i.e. there will be multiple levels where leaf nodes reside. This provides for non constant density along the dimension.

In order to scale these functions vertically, a vector $\vec{\alpha}$ is used to store vertical scaling coefficients. When this is applied to higher dimensional space, the basis functions are calculated as a tensor product of their first-order functions:

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \varphi_{l_j, i_j}(x_j).$$  \hspace{1cm} (3.3)

$\vec{l}, \vec{i}$ constitute the level-index vector, indicating the level and index for each dimension. See Figure 3.1 for a demonstration of how a point in space is addressed using level index vectors. To describe the position of a grid point in one dimension, it is sufficient to state its level from the root node and its index from the origin of the dimension.

Common basis functions include the linear hat function examined here and the B-spline basis function. Pflüger [5] examines these and other basis functions.

3.3. The Subspace Scheme

One method to generate a regular sparse grid is the subspace scheme. It utilizes multiple different subspaces to composite a sparse grid. Such a subspace is created by using the
3.4. Function interpolation

Using a linear combination of these basis functions it is possible to approximate a function \( f(\vec{x}) \) using sampling based techniques to arrive at \( u(\vec{x}) \)

\[
f(\vec{x}) \approx u(\vec{x}) = \sum_{|\vec{l}| \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}).
\] (3.8)

3.4. Function interpolation

Using a linear combination of these basis functions it is possible to approximate a function \( f(\vec{x}) \) using sampling based techniques to arrive at \( u(\vec{x}) \)

\[
f(\vec{x}) \approx u(\vec{x}) = \sum_{|\vec{l}| \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}).
\] (3.8)

Figure 3.2.: Constructing a sparse grid from several subspaces. If the greyed-out grids are added as well, the result would be identical to a full grid. Figure courtesy of [5].

linear span of the basis functions

\[ W_{\vec{l}} := \text{span}\{ \varphi_{\vec{l},\vec{i}}(x_j), \vec{i} \in I_{\vec{l}} \}. \] (3.4)

The indexes are pulled from the generalized index set \( I_{\vec{l}} \),

\[ I_{\vec{l}} := \{ \vec{i} : 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ odd}, 1 \leq j \leq d \}. \] (3.5)

To combine these into the sparse grid \( V_n \), where \( n \) is the level of the grid, every subspace with

\[ l_j \leq n + d - 1 \] (3.6)

is combined into a sparse grid in the following manner:

\[ V_n^{(1)} := \bigoplus_{\sum_{j=1}^d l_j \leq n+d-1} W_{\vec{l}}. \] (3.7)

See Figure 3.2 for how this works graphically. This, as well as a more detailed explanation of the scheme can be found in [5] and [10].

3.4. Function interpolation

Using a linear combination of these basis functions it is possible to approximate a function \( f(\vec{x}) \) using sampling based techniques to arrive at \( u(\vec{x}) \)

\[
f(\vec{x}) \approx u(\vec{x}) = \sum_{|\vec{l}| \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}).
\] (3.8)
3. Sparse Grids

Figure 3.3.: A linear combination of the linear hat basis function to approximate a function $f(x)$. The left side illustrates the combination of the basis functions to interpolate $f(x)$, the right side shows the separate basis functions $\varphi_{l,i}$ weighted with their $\alpha_{l,i}$ coefficient. Figure as seen in [5].

See Figure 3.3 for how interpolating a function works with hierarchical basis functions in the one dimensional case.

3.5. Density Estimation

Density estimation is the practice of using a data set to estimate a hidden probability density function (PDF). One approach is to use kernel based density estimation [11]. Kernel based density estimation uses a linear combination of so called kernel functions centered around the training points to approximate the function.

Another approach is to use a variational approach [6],

$$\hat{f} \approx \arg\min_{f \in V} \int_{\Omega} (f(\vec{x}) - f_{\varepsilon}(\vec{x}))^2 d\vec{x} + \lambda ||\Lambda f||_{L^2}^2. \quad (3.9)$$

$V$ is the function space to search for $f$. $\lambda ||\Lambda f||_{L^2}^2$ is used for regularization of the resulting function. It provides for the ability to adjust the function’s sensitivity towards each training point.

This can be transformed into a system of linear equations, see [6],

$$(R + \lambda C)\vec{\alpha} = \vec{b}. \quad (3.10)$$

$\lambda$ is the regularization parameter used to smooth the resulting probability density function. Matrix $C$ is the regularization matrix. For simplicity, it is possible to set this to the identity matrix $I$, if the regularization is to be applied uniformly [6].

Using this, the system matrix can be simplified to

$$R + \lambda I. \quad (3.11)$$
3.6. Spatial Adaptivity

Previously, the technique of spatially adaptive refinement was decomposed into two tasks. The first was to detect where to refine next. Multiple of these algorithms to detect “interesting” regions, so called refinement indicators, exist. Surplus based refinement measures the magnitude of the grid point’s entry in the $\vec{\alpha}$. A higher value indicates a region where the probability density function has high variance, indicating an “interesting” region. Other refinement indicators, such as the zero-crossing refinement functor or the data based refinement functor can also be used. For classification, the actual method of refinement should not affect the logic of the learning phase itself.

The second task is to insert, or potentially even remove, grid points where necessary. When points are inserted, they are inserted into a lower level of the hierarchy. However, some vertices on the same level with another index might need to be inserted as well, as seen in Figure 3.4. As this operation changes the grid, several variables such as the system matrix or the $\vec{\alpha}$ vector need to be adjusted to fit the new grid.

3.7. The Cholesky Decomposition

To solve Equation 3.10, a decomposition can be used. This allows the system to be solved for different $\vec{b}$ without the need to solve the entire system again. One of these available decompositions is the Cholesky decomposition. It works when the system matrix is positive semi-definite. One advantage of the Cholesky decomposition is that it supports refinement without a need to recalculate the decomposition, a step that can be computationally expensive for a high number of grid points. The current implementation also supports an incomplete decomposition, which lowers the accuracy slightly by approximating the
3. Sparse Grids

decomposition but causes a high speedup in exchange.

The implementation of the parallel learner will only support the Cholesky decomposition for the start. The Cholesky decomposition was implemented by Adrian Sieler in [11]. Consisting of two steps, the decomposition has an offline decomposition step and an online decomposition step. The offline decomposition step is a decomposition that is run before training commences. This is the step of finding $L$, such that

$$LL^T = \mathbf{R} + \lambda \mathbf{I},$$

(3.12)

see Section 5.1 of [11] for its inner workings. For now, it shall be assumed that the decomposition is already available when training starts, obtained by loading it from the file system. Additionally, there is a modification step that is necessary every time the grid changes, adding or removing basis functions, which is examined in Section 6.5 and elaborated in [11]. This modification step is important to consider in the parallelization, as the results need to be updated on all nodes in a computationally inexpensive manner. The implementation should also attempt to be as general as possible, to allow for eventually switching the Cholesky decomposition for other decompositions.
4. Parallelization

4.1. Existing solutions

Certain aspects of the creation of sparse grids can be parallelized in order to decrease the
time to solution. The first approach examined by Pflüger is to use OpenMP to parallelize
specific functions using the UpDown scheme [5]. One of the first issues with parallelizing
in this manner is that the results were no longer reproducible, as the magnitude of the
rounding error was dependent on the order of the summation of results. Fortunately, the
error remains so tiny that these fluctuations are nearly irrelevant, however the system is
no longer entirely deterministic. Heinecke shows empirically that although the recursive
scheme presented by Pflüger [5] has a better asymptotic runtime than working on the grid
directly, the direct method can be optimized in a way that takes advantage of hardware
to decrease the time to solution manifold [4]. Specifically, the original implementation
suffers from creating a multitude of cache misses, as it accessed the data structure in
a very non-sequential way. The second solution, however, works on each grid point
sequentially, taking advantage of caching at processor level. To further improve upon the
algorithm, branching statements could be moved as far out of loop as possible, such that
the processors branch prediction would create fewer misses and eliminate a majority of the
required rollbacks. Peherstorfer also proposed a method to split the computation process
into an offline and an online part [6]. The offline part adds a preprocessing part that
decomposes the system matrix to speed solving the system of linear equations. As soon
as data is available, the online algorithm can then solve the system of linear equations
in $O(n^2)$ instead of $O(n^3)$. These previous parallelizations were all designed for general
purpose calculations with sparse grids. Furthermore, these parallelizations are mostly
node-based, using for example OpenMP parallel pragmas. These can be parallelization
across one system, but not over multiple systems. This thesis, on the other hand, concerns
itself with the parallelization of a very specific use case, the training phase of a classifier,
in a way that allows it to be parallelized not only across the one system, but across many
networked systems in a cluster-based parallelization. [6].

4.2. Parallelization approaches for the new learner

Building on the previous work in parallelization, some of which is used by the current
learner, the new parallel learner can be implemented. Several different approaches exist to
divide the entire task of learning the probability density functions into the smaller tasks.
4. Parallelization

4.2.1. Classes

One available option is to parallelize over the different classes present in the data set. Every node can be used to learn the probability density function of its class independently of all the other classes. In the end, all these functions are collected into a classifier.

**Advantages**  This method of parallelization requires very little communication and only requires a few implementation changes. Each node only needs to hold the grid for its own class in memory, saving space. It is also highly efficient under the assumption that the test set contains roughly equal data points for all classes.

**Disadvantages**  While there are data sets that do fulfill the requirement of balance between classes, there are also those that do not. If one class has been sampled much more often than another because they are not equally probable, then work will be split unevenly. With such data sets that have a majority of samples from one class this parallelization scheme will not perform much better than the sequential version, as the node with the most data points will slow the time to solution. Such an example can be seen in Figure 4.1. Another severe restraint is that the potential to parallelize is restrained by the number of classes. With datasets of size 2, such as the Ripley Garcke or Chessboard datasets, the maximum speedup would also be 2.

4.2.2. Spatial

Another option is to parallelize over the problem (grid) space. While this technique is useful in other regions of high-performance computing, several issues prevent its use in this scenario. On one hand there is the non-uniform nature of the sparse grid, which is much denser in some regions than it is in others if adaptive refinement is used. On the other, there is again the likely non-uniform distribution of data samples to learn from, which will also be more dense in certain regions than in others. An extreme example can again be seen in Figure 4.1, where the spatial splitting of data points onto nodes causes node 1 to receive regions of very low density. While it is possible to compensate for these, this approach was dropped in favor of batch parallelization.

4.2.3. Batches

The idea of batch based parallelization is to split the training set into multiple smaller batches of specific size. A node learns from its batch by computing the density functions of all the classes and later combining the results with the other nodes.

**Advantages**  This approach is fairly flexible and can scale to adjust to diverse computational capacity on different nodes. It does not suffer performance penalties when the class sizes are unequal. A good scheduling algorithm will be able to distribute computational tasks such that all processors are utilized to their capacity.
4.2. Parallelization approaches for the new learner

Figure 4.1.: Problems with class-based and spatial parallelization. In the case of class-based parallelization, only the very few points in class 1 (orange) exist, all of them being assigned to node 1. Meanwhile, a large majority of the points are in class 0 (blue), and are therefore assigned to node 0. In the case of spatial parallelization the same effect occurs, when the space is subdivided into 5 segments along each dimension. While this is unlikely to occur in such a drastic manner, regions of differing densities will have a performance impact. In both cases node 0 has a much higher workload, which in turn increases the time to solution as it slows down the node on the critical path.
4. Parallelization

Disadvantages This approach requires a lot of communication, as batches need to be assigned, results need to be communicated and grids need to be transferred after refinement. It also requires each node to hold all grids in memory in order to be able to learn for all classes.

4.2.4. Evaluation

For the new learner, the batch based parallelization scheme was chosen, as the high flexibility was deemed worth the extra complexity in communication and the additional memory requirements. This type of parallelization can be categorized into coarse-grained parallelism. The approach can not be classified as embarrassingly parallel, as grid changes and \( \alpha \) vectors still need to be transferred, but neither is it fine-grained parallelism, as large batches do not require many communication requests per second. This is a type of task parallelism, as different nodes can be executing entirely different tasks at any point in time. In Flynn’s taxonomy, this approach falls into the *Multiple instruction streams, multiple data streams (MIMD)* category. This is the most common type of parallelization category in the taxonomy [12].

4.3. State of the Learner

To implement the new learner, the current learner can be analyzed and subsequently refactored. By increasing the modularity through refactoring coupling is decreased, allowing for parts of the program to be outsourced to the different nodes. To start out, the current implementation is decomposed in Figure 4.2. The aim is to distribute the available tasks onto a heterogeneous processing environment while minimizing the time to solution. Although the process of classification has multiple computation intensive segments (such as the offline Cholesky decomposition, the refinement, or the prediction of large actual data sets), parallelizing the training phase will be the primary focus of this thesis.

4.4. Distribution of Roles

The parallel implementation will have two roles, the master node responsible for communication and refinement as well as several worker nodes.

**Master Node** The master node is responsible for distributing batches for the workers to learn from. While the workers are busy with learning, the master uses the currently available alpha vector and does one refinement step for all classes. As soon as the refinement completes, the master distributes the updates to the workers and receives new alpha values from them.
4.4. Distribution of Roles

1: Load dataset meta data (Dimensionality $d$, #Classes $k$) and data
2: Create initial full grid
3: Find regularization parameter $\lambda$ by cross validation
4: Construct system matrix with $d, l_s, \lambda$
5: for Batch of data $j$ do
6: Load batch $j$ of the training data into memory.
7: Separate batch into $k$ class-batches, one for each class
8: for Each class-batch $i \in \{0, ..., k\}$ do
9: Calculate class specific density vector $\alpha_{j,k}$
10: Combine $\alpha_{j,k}$ with $\alpha_{j-1,k}$ from previous corresponding class-batch weighting accordingly
11: end for
12: if a refinement is scheduled for this batch then
13: Refine the grids of all classes
14: end if
15: end for
16: Finalize training
17: Begin classification
18: for Each point $p$ to classify do
19: Find $a_k$ with the highest value at point $p$, considering potential class weighting factor
20: Classify point $p$ into class $k$
21: end for

Figure 4.2.: The current implementation of the Batch Learner. This shows the full problem solution from start to finish in simplified manner. This includes preparation steps on lines 1-4, a section on training in lines 5-16, and the actual prediction on lines 17-21. The goal of this thesis is to parallelize the training step, on lines 5-16. This includes attempting to handle the sequential refinement algorithm while continuing to learn in parallel.
4. Parallelization

**Worker Node**  The worker node has two distinct responsibilities. The first is to complete any learning requests by the master and subsequently hand in the results. The second is to update the system matrix after refinement has been conducted by the master and submit the results for broadcasting to all other workers.

The difference between these two roles may be seen in the pseudo-code of Figure 5.1. Although their tasks differ, the incoming requests are all processed in one routine for incoming messages to avoid duplication, since some communication routines overlap between the master and the client.

4.5. Refinement

One of the long-running computations, save for the actual batch learning, is the refinement. The refinement uses the current density functions to add or remove grid points from the grid. For now, the process of refinement will be viewed as a black box that will be running on a single node (leaving open the possibility that the functors themselves can eventually be parallelized). As far as this thesis is concerned, the refinement is a sequential algorithm which can be called that will eventually return the refined grid. From the refined grid, a list of grid points that were added/removed is extracted. The deleted grid points list is returned directly by the refinement functor, referencing the old indexes of grid points that no longer exist. The list of added grid points can be extracted manually by checking the difference in size between the old grid and the new, and examining that many points at the end of the data structure, which will correspond to all new points in the grid. Using this information, the alpha vector and the Cholesky decomposition is adjusted to fit the new grid. In the current implementation, the program will now continue learning a few batches using the new grid before refining further.

However, it is not desirable to let all nodes idle during the time that one node calculates the refinement step. Instead, the other nodes should continue learning batches with the old grid until the refinement completes. Then, once the refinement is done, the refined grid is distributed to the other nodes and each node merges the changes into its grids in between learning batches.

4.6. MPI-Communication

The Message Passing Interface (MPI) is an open parallelization standard based on multiple processes working independently and passing each other messages. This standard was implemented by multiple vendors, among them OpenMPI [13] and MPICH [14], both of which were used during the implementation of the learner. Fortunately, the standard allows vendor independent programs to be written, allowing the user to compile against an MPI implementation of their choice.

A node is identified by its MPI rank, an unsigned index starting from zero unique to every node (process). The learner uses the MPI rank to determine whether a node is a master, equivalent to rank 0, or a worker, any other rank. Up to the point where a
program performs conditional behavior based on its current rank, all nodes follow exactly
the same workflow. In Figure 5.3, this is exactly the point where the training method is
called by LearnerSGDEOnOffTest or by the end user. The first thing a process determines
once training starts is its role, and launching the appropriate method based on the result.

Another important MPI feature is the ability to determine the world size. This is
primarily used by the master process to determine how many workers it can communicate
to. Among other things, this is especially important for the scheduling algorithm; the
algorithm needs to know the number of workers to distribute the requested tasks fairly.

Asynchronous send and broadcast  MPI offers many different communication tech-
niques to make parallelizing as straightforward as possible and hide internal complexity
from the developer. For all the potential communication diversity, the learner relies
heavily on only two of these communication routines, ISend and IBcast. ISend is used for
asynchronous one-to-one communication and IBcast for asynchronous one-to-all (broadcast)
communication (which is only initiated by the master).

Anytime one of the nodes sends a message, it is not important for that node to know
whether the message has been received successfully or even whether there will be an
immediate response. Instead, it is desirable for the node to continue working after a
communication request has been submitted even when the request has not been sent or
delivered yet. MPI asynchronous operations, which include ISend and IBcast, allow a node
to continue working while MPI facilitates the communication requests in the background.
However, each node needs to ensure that the buffer used to send or receive data is not
modified or reused until MPI confirms that the request is completed using Status or Wait.
This is relevant for Section 6.1.

MPI broadcast mechanisms are designed such that the implementation can be delin-
earized. Instead of having the original node send the data to each client separately, all
the data could for example be distributed in a tree model (each non-root non-leaf node
receives data once and sends it to n other processes). This leads to the benefit that the
total overhead only grows logarithmically with respect to the participating nodes instead
of linearly. Additionally, such an implementation allows the communication overhead to
be distributed across more nodes than just the root. One minor constraint is that every
node needs to know the root node, where the data originates, before joining a broadcast
request. The current learner ensures this by only allowing broadcast requests from the
master to all workers.
5. The Implementation

For important aspects of the inner workings of the learner and especially how communication is handled, see the pseudo code in Figure 5.1. It shows the difference between the master’s and the worker’s workflow, and how they use the same routine to handle any incoming requests. While the master handles multiple jobs, the worker only ever reacts to an incoming request.

Figure 5.2 shows an example of events during a training cycle. It shows the assignment, training, update and merge phases during the system lifecycle.

The architecture of the implementation is split up into several diagrams. Figure 5.3 shows the main parts of the learner, with the actual learning logic implemented in LearnerSGDEOnOffParallel. MPIMethods manages communication with other nodes. LearnerRefinementHandler contains logic extracted from the original learner that takes care of the refinement process itself.

Figure 5.4 shows the different types of messages used throughout the learner to communicate with other nodes. Their purpose is explained in Chapter 6.

The task scheduling as well as miscellaneous containers are found in Figure 5.5. These are also examined later in Chapter 6.

5.1. Implementation Details

Over the course of time spent coding, approximately 2.5 months, the entire learner was implemented in a largely self-contained manner in just under 20 C++ types. An effort was made to ensure that the MPI implementations are reusable by future extensions to the learner with as little modification as possible. Relevant parts of the application were unit/integration tested using the Boost [15] and MPI frameworks [13], [14]. The Doxygen [16] framework was used for documentation.

5.2. Implementation Challenges

Refactoring  One challenge encountered in the implementation of the parallel learner was that the original learner was not exceedingly modular. In fact, almost the entire learning algorithm was implemented in one lengthy method named `train`. During analysis, this method was decomposed into many smaller methods during refactoring. Only this hiding of complexity helped comprehend exactly how the original learner actually worked. Meaningless variable naming or very complex structures were another challenge encountered during the refactoring.
5. The Implementation

Master
1: Load dataset
2: Grid setup
3: while end of training not reached do
4: Process incoming MPI requests
5: Assign batch to a worker
6: if Refinement is scheduled then
7: Check old batch work complete (6.3)
8: Call refinement functor
9: Send changes (6.4)
10: Assign Cholesky updates (6.5)
11: Wait for Cholesky updates (6.5)
12: Rebroadcast Cholesky updates (6.5)
end if
end while

Worker
1: Load dataset
2: Grid setup
3: while no shutdown signal received do
4: Process incoming MPI requests
5: end while

Process incoming MPI request (for both worker and master)
1: Pull top message from incoming message stack
2: switch Type of message do
3: case Assign Batch (worker)
4: Assemble Batch
5: Train from Batch
6: Send α values (6.3)
7: case Compute Cholesky Factorization (worker) (6.5)
8: Wait for refinement results to arrive
9: Compute Cholesky modification
10: Send Cholesky modification
11: case Receive refinement results
12: switch Result Type do
13: case Additions (worker) (6.4)
14: case Deletions (worker) (6.4)
15: case Cholesky (master/worker) (6.5)
16: Adjust matrix size
17: Copy data to offset
18: case Receive α values (master) (6.3)
19: Adjust received vector for current vector
20: Merge α vectors
end switch
end case
end switch

Figure 5.1.: Parts of the Batch Learner. A worker spends its entire life cycle acting on, and responding to, incoming MPI requests until the connection is severed. The master on the other hand additionally handles assigning batches and all work concerning refinement. Numbers in brackets are references to the section where they are explained in detail.
5.2. Implementation Challenges

Figure 5.2: Sample sequence of a running parallel batch learner. During the learning loop, workers are each assigned several batches to train from. At the end of each batch, the worker sends the current \( \alpha \) vector for each class to the master process, where it is merged into the collective alpha vector. As soon as data has arrived, the master starts a refinement cycle during which the workers keep learning. The grid changes are then broadcast, such that the workers can update their grid in the next cycle. Each class now needs an update of the system matrix. These computing tasks are also distributed to the workers, which hand in their results separately. Once all the new system matrices have been transferred, the next learn cycle may begin.
Figure 5.3.: Class diagram of the current learner and the MPI container. Request tracking and MPI message data may be found in Figure 5.4. Figure 5.5 shows the scheduling and auxiliary structures. Note that only the more important functions are shown for brevity. Several functions for internal use were omitted.
5.2. Implementation Challenges

Figure 5.4: Class diagram of the MPI message data containers and the pending request tracking system. The MPI Request Pool is covered in Section 6.1. Network message data is described in Sections 6.3, 6.5, 6.4.
5. The Implementation

Figure 5.5: Class diagram of the scheduling and the auxiliary structures. Currently, a round-robin scheduler is implemented, but the callbacks for the scheduler to observe system state should allow for more complex scheduling behavior, where workload can be tracked on a per-node basis. Auxiliary structures are used throughout the entire subsystem as containers for the transfer of data.
5.2. Implementation Challenges

**Update Cholesky Decomposition**  Before the Cholesky update was parallelized itself, every node would still compute its own update. If there happened to be multiple updates in a row without any learning, through segmentation of large packets for example, the updating of the Cholesky factorization would cause a segmentation fault allocating data matrices, something semantically completely unrelated.

**Reinterpreting Bit Patterns**  Similar to the unset command ids from above, a lot of the communication code involves interpreting memory regions based on their contents. Faults in both the reading and the writing could go unnoticed for a long time and cause problems in unrelated sections of the program. This communication section especially required a lot of manual and automatic testing to find individual problems that caused undesired results.

**Sequencing**  Unfortunately, using both *IBcast* and *ISend* meant that there was no more guarantee that packets would arrive in the order they were sent as a broadcast could complete faster than a send or vice versa. This meant that a series of requests, such as applying grid changes, could suddenly be broken by a completely unrelated request, such as a learning request. This caused undefined behavior due to an inconsistent state. To compensate for this, extensive tracking requests and state tracking was implemented, so that such situations were detected reliably and necessary action could be taken to ensure the sequencing was restored to the originally intended. To do this, for every packet type preconditions and postconditions were identified and implemented using predicate callbacks.

**Tracking Requests**  After having determined that packets needed to be checked before they are processed, a packet tracking request was added to check these conditions. Such a tracking request has a callback that is executed on every packet until the tracking request deregisters itself. Using these, the processing of a packet could now be interrupted to process other packets, eventually returning to the original process once a specified system state has been reached.

**5.2.1. Communication**  

Already, the Pseudo-Code in Figure 5.1, the Sequence Diagram in 5.2 and the MPI Communication Class Diagram in 5.4 have shown the overall structure of the MPI Communication, which will be elaborated in the following chapter.
6. Communication-Plan

6.1. MPI Request Pool

As previously mentioned, when using asynchronous requests it is the callers responsibility to ensure the buffer is protected until the request completes fully. In this case, more memory is used to store the pending requests in an attempt to decrease IO idle time. To manage these requests, a pool of resources has been implemented that manages allocated buffers, handles callbacks for completed requests and stores handles to the pending MPI requests.

One may desire to use the MPI routine \texttt{MPI\_WaitAny} to allow idling until one of the requested operations completes. This method allows the handing in of an array of pending MPI requests and returns the index of the request that completed. Whether this idling is performed using busy wait, as can be observed in OpenMPI, or interrupt mechanisms, is implementation dependent. This routine requires all the MPI requests to be held in memory sequentially. However, for every pending MPI request, the buffer, as well as the callback and minor additional information needs to be stored. These, however, need to be stored separately and may only retain an index of where the corresponding MPI request is located, as seen in Figure 6.1. This index is not a pointer, as the MPI request storage can move in memory when the storage grows. Instead it is storage as an index from the start of the vector. These are the references from the additional request information storage to the MPI request storage in Figure 6.1. A second vector stores indices to MPI Requests that have already been completed, that can be re-used. Since the index of an MPI Request cannot change (this would invalidate the index reference from the additional request information), these cannot simply be deleted.

From this reasoning, 4 functional requirements can be extracted that the MPI Request Pool needs to fulfill.

1. MPI Requests need to be held in memory sequentially for \texttt{WaitAny}.
2. Additional request information needs to be associated to the request.
3. The pool needs to facilitate growing when adding and shrinking when deleting for differing communication intensities.
4. No pointers may be used when storing data in vectors (they can move in memory upon resizing).

The current implementation of the MPI Request Pool, sample state seen in Figure 6.1, fulfills all the above requirements.
Figure 6.1.: Sample state of an MPI Request Pool. The lookup for an MPI_Request given the additional request information can be completed with asymptotic complexity in $O(1)$, meanwhile the inverse lookup from an MPI_Request to the additional request storage is linear with the additional request storage vector size.
Figure 6.2.: Allocating operation on the MPI Request Pool. Since the MPI Request Pool has open slots available for using, the first free request in the storage vector is reused after storing the additional request information in the addition request information list. The list storing the available requests is then shortened by the handle that was just used.
Figure 6.3.: Deleting operation on the MPI Request Pool. First, the MPI_Request and the additional information section is deleted (#1). If the deleted request was situated at the end of the storage, the end of the available requests vector is searched for whether the vector can be shortened further than just the current element. If this is the case, the MPI_Request Storage is shortened all the way up to the last non-free slot (#2).
6.2. The Scheduler

Allocating  Increasing the pool size is relatively straightforward. If there are available request slots, one of these is used to avoid resizing the potentially large MPI Request Storage vector. In this case, only the additional request information needs to be resized and given a reference to the new element. If there is no free space in the MPI request storage vector, the MPI request storage vector as well as the additional request information vector are resized to fit more elements and the reference is then assigned. Note that during this vector growing operation, the vector may realize there is no free memory in its currently allocated slot. In this case, the vector copies its data to a new memory region of higher size. Fortunately, this occurs in amortized constant time, see [17] for details.

Deleting  Deleting a request normally only shrinks the additional request information vector and adds the index to the available requests. However, assume the last element (number 5 in Figure 6.1) is now deleted. Previously, positions 3 and 4 in the request storage could not be deleted as the index of request 5 was not allowed to change (it was referenced by additional request information 2). Now, indexes 3, 4, and 5 in the storage are all free requests. This allows the storage to shrink back to size 2, freeing memory as the communication queue becomes smaller. The appropriate indexes in the available requests are used to track previously freed requests, so that the pool can know how many indexes from the back of the vector it is permitted to delete. Once the request has been deleted, the corresponding available request is disposed of as well.

6.2. The Scheduler

The scheduler is responsible for assigning work to the workers. It currently has several hooks available to it for assigning work and keeping track of work. When ever the master node discovers there is work to be done, it solicits the MPI task scheduler implementation for which node should do the work and, if applicable for the task, how large of a task should be assigned. The available tasks are learning from batch, updating the system matrix, and refining.

Round Robin Scheduling  The current implementation sports a round-robin scheduler, which is, if not very smart, at least straightforward and effective. The round-robin scheduler assigns work to workers in ascending order according to the MPI Rank. It uses constant sizes for assigning batch training sessions, equal to the batch size. It also keeps track of pending learning requests, in order to schedule the refinement cycle accordingly (see 6.3 as to why this is necessary).

Future work  A smarter scheduling algorithm could be implemented in the future in order to further increase usage of the available processing time on all nodes. Possibilities include varying batch sizes (larger batches for nodes that will not receive system matrix update requests) and offloading some work onto the master node itself.
6. Communication-Plan

6.3. Alpha Merge Request

The alpha merge request is triggered whenever a worker finishes learning from a batch. The worker takes its alpha vector and splits it into chunks that fit into the specified MPI Packet size. Each packet is outfitted with the class index, the batch offset and size it is trained from, as well as the total unsegmented size of the alpha vector. The total size is used for allocation of the vector on the master and for determining whether all packets have successfully arrived. It also specifies the grid version used to learn the alpha vector, this is used by the master to detect when its version of the alpha vector no longer corresponds to the one the batch has been trained on. When the master receives the alpha vector and the grid versions still match, the master can then merge the vector, weighted if necessary, with its own alpha vector.

Grid-Version mismatch  If the master has completed a refinement cycle before the merge request arrives, the grid versions, and thus the alpha vector lengths, will no longer be equal. One option would be to ignore the results and continue learning, or to re-learn the specific batch with the newer grid. The current parallel implementation uses another option to recover from this situation inexpensively. Since the refinement results are still stored in memory until the start of the next refinement, the changes from the last refinement result can be used to update the alpha vector by one version by deleting the corresponding deleted values and expanding the vector by the number of added grid values. Now, the alpha vectors have the same size and can be merged as if they were trained from the same grid.

Requirements to the Scheduler  One problem that can occur is that by the time the alpha merge request is processed by the master, more than one refinement cycle has already passed (the grid versions differ by at least two). If coarsening is disabled, it is still possible to recover by just adjusting the size of the alpha vector (however, this is currently not implemented). If coarsening is enabled, it is no longer possible to adjust the vector, as the corresponding refinement results have already been overwritten (unless they are backed up in a different location). A semi-smart learner can avoid this situation by keeping track how many batch assignments are pending for the current and the last grid version. As long as there are still pending assignments for the last grid version, the scheduler can prevent the master from refinement, instead forcing it to keep processing incoming requests. This effectively prevents a second degree version mismatch from taking place, incurring only a minor performance penalty. In the future, this would be the point in time when the master decides to perform learning itself. It is possible to deactivate this feature in the scheduler, transferring the requirement to prevent second level mismatch to the user, who can adjust the input parameters to make the mismatch very unlikely. If such a mismatch does occur, the results from that particular batch will be discarded.

Further improvements  If, in the future, it is desirable to use compression to reduce network load from these requests, the alpha values could be transferred as compressed
delta values to reduce the size of communication requests to sub-linear with respect to the number of grid points. Whether this potential improvement is worth the cost of implementation, however, is subject to further evaluation.

**Complexity** Each class has a separate $\vec{\alpha}$ vector that needs to be transferred to the master. Each $\vec{\alpha}$ vector has length equal to the number of grid points. Thus, the communication complexity lies in $O(|K| \times |G|)$. Over the course of the entire learning process, the number of merge requests is equal to the number of batches $|B|$, however the number of grid points is not constant during the learning process. The complexity of all requests during a learn cycle can be written as $O(|K| \times \max(|G|) \times |B|)$.

### 6.4. The Grid Update Request

The grid update request is sent by the master whenever a refinement cycle is completed. There are three types, one for deleted grid points, one for new grid points, and one for the updated Cholesky decomposition. All of these requests are automatically segmented to prevent large memory allocations, especially relevant for the transfer of a potentially large Cholesky decomposition.

**Additions** Originally, each grid point was identified by its coordinate along each axis of space. This was convenient, as the method is intuitive and the complexity increases only linearly with respect to the number of dimensions. However, extracting and inserting grid points this way was not as straightforward, as points would have to be located in the hierarchy based on their position in space. Instead, the implementation was replaced transferring the internal coordinate in hierarchy space. Now, each grid point is uniquely identified by $d$ different level-index pairs, stored in the level-index vector (as was previously examined in Figure 3.1). Upon reception by the worker, the worker can create a new grid point directly from the level index vector and insert it into the grid.

**Deletions** The storage indexes of deleted grid points from coarsening are also transmitted. These are very inexpensive to transfer, as all that is required to identify them is their position from the start of the grid (which is stored as a vector). The grid points can then be removed easily as long as they are removed in order, as every removal affects all indexes behind the removed point. Thus, for the deletion operation it is important to ensure that all points are copied, sent, received, and applied conserving their order.

**Updates and Alpha vector** After both the grid additions and deletions have been received and the learner has detected that the transfer of refinement results is complete, these refinement results are then applied in one go (even though they might have arrived heavily segmented). After the grid itself is updated, the entire grid's leaf property (whether a grid point does not have any children) is recalculated. This is currently a substitute for a smarter way to update the leaf property on just the points that need them. One
potential such implementation could include finding the parent in every dimension of a just inserted grid point, and setting their leaf property to zero. After the grid itself is updated, the refinement results are also used to shrink and then grow the alpha vector, such that each entry still corresponds with a grid point.

**Complexity** The grid update only transfers changed grid points, where deleted points are especially cheap, requiring only one index transfer ($O(1)$, per deleted grid point). The addition of a single point is slightly more expensive, as two coordinates need to be transferred per dimension ($O(d)$). The combined communication complexity of an entire merge request is in $O(r + ad)$, with $r$ deleted points, $a$ additions and dimensionality $d$. Thus, grid merges are linear with the number of changes if the dimensionality remains constant.

### 6.5. System Matrix Update

After every refinement, the Cholesky decomposition needs to be updated. As this is one of the tasks that will potentially use a lot of processing time when the grid is large, this update step is also parallelized. The update of the decomposition is parallelized by class, as the decompositions are dependent only on their class. Using the scheduler, the update Cholesky requests are distributed from the master. Every worker ensures their grid refinement has completed, and then computes the update of the decomposition. The entire decomposition is then sent to the master, which then rebroadcasts it back to all the clients. This transfer of the decomposition is expensive ($O(|G|^2)$, with $|G|$ grid points) and it is heavily segmented into packets to reduce large allocations, but causing significant overhead. In the future, it will be desirable to find potential methods to reduce the size of the transferred results. The rebroadcasting from master is required as the asynchronous broadcasts need a root node, which needs to be known ahead of time. For one worker, this causes the results it has computed to be sent back to it, wasting some of its available processing time. However, this broadcast should be more efficient than a linear sequence of point-to-point communications, as it allows the MPI framework to use non-linear transfer of data. While this is not an unefficient implementation, a potential solution that is likely to be superior involves the master sending a broadcast message telling the workers to listen to a broadcast from that worker specifically, so the entire transfer can be done in one MPI communication rather than two. However, this approach would require additional checks to ensure messages are processed in the correct order, as the master can no longer order them correctly itself.
7. Analysis

In order to have a meaningful comparison between the old learner and the new, several performance tests will be executed. The goal is to measure the overall efficiency of the parallel implementation and compare its performance to the original learner.

7.1. Classification Accuracy

To show the new implementation is competitive, a demonstration of the accuracy will be performed.

Aim To examine the effect of learner parallelization on the accuracy of the classification result.

Hypothesis As mentioned earlier, the classification accuracy is expected to drop slightly in comparison to the original classifier. In addition, the variance of the classification results is expected to increase. This is due to the nature of asynchronous learning, as one might get lucky with timings, raising the classification accuracy, or unlucky. Since timing is not an issue in synchronous learning, the accuracy is expected to fluctuate less.

Methodology The tests will be conducted with the original learner, as well as the new learner with world sizes 2, 4, 6, and 8. At the time of writing, the new learner did not yet support learning with a world size of 1. The first data set in use will be the Ripley-Garcke dataset, with one refinement cycle and a batch size of 5 data points. Each learner will be trained and tested 50 times.

Results See Figure 7.1 for a Box-and-Whisker plot of the results. From these tests, it appears that there is very little, if any, performance decline from the original learner. In fact, the worst result observed from the new learner is only 0.2% lower than the worst result observed on the original learner. The median shows the same trend of only very slightly worse results. Both the inter-quartile range and the total range have not increased significantly, for some trials they have instead decreased.

Conclusion From this experiment it may be observed that, most importantly, the negative influence of the parallelization has remained insignificant enough to be barely noticeable. Furthermore, the spread of the classification accuracy can not be correlated to the number of workers in the job.
7. Analysis

![Classification Accuracy on Ripley Garcke Dataset](image)

Figure 7.1.: Classification accuracy for different learners on the Ripley-Garcke dataset. The results are shown in a Box-and-Whisker Plot. From top to bottom, the regions stretch from the maximum to the upper quartile (top whisker), from the upper quartile to the median (orange box), from the median to the lower quartile (gray box), and from the lower quartile to the minimum (bottom whisker).

One might question why the original learner does not have deterministic results (range 0) when presented with the same parameters. Inspection shows that small parts of the training were already previously parallelized using OpenMP. Whether this is the true origin of the spread, is currently unknown. However, more important is the conjecture that the parallelization has not significantly contributed to the variance of classification.

**Evaluation** In order to improve the precision of the results, a different data set could be used with a larger test set. The Ripley-Garcke data set only contains 1000 test values, providing a less than stellar 0.1% resolution.

7.2. Tuning

In order to understand what variables have a significant impact on performance, a three worker one master setup will be examined and then tuned to minimize the time to solution.

**Applicability of the learner** It is important to understand that it does not make sense to apply the parallelized learner in its current state to all potential classification problems. Whether the learner will actually reduce the time to solution depends on some variables
7.2. Tuning

![Classification Time on HEPMASS Dataset](image)

Figure 7.2.: Unfortunate performance results of the untuned learner with the HEPMASS dataset. When the learner parameters are set in a way that hinders the parallelization to be effective, the time to solution might actually increase.

examined here and multiple variables that were not directly measured, such as time spent loading the dataset into every node. Recognize that the measurements performed in the following experiments measure only the training time itself, and therefore are only indicative of whether using the parallel learner over the sequential is to be preferred in a specific situation.

**Untuned Performance**  A scenario that shows particularly bad performance can be seen in Figure 7.2. In this case, the parallelized learner performs 19.3% worse on three nodes than the non-parallelized learner does on one node. This is due to several factors examined in the upcoming section.

### 7.2.1. Idle Times

One of the largest performance impacts on the current run is the idle time. The master idles whenever it is waiting for the workers to complete jobs before adding new items to the working queue. The workers idle when they run out of jobs and the master is busy, thus not assigning them new jobs. The level of idling can be seen in Figure 7.3.

Idling is especially harmful on the OpenMPI Implementation, as waiting for messages is implemented with spin loops using 100% of the available processing power on that core. Normally, this would not be much of an issue, however several unrelated parts the learner are also parallelized using OpenMP to run on multiple cores. Since OpenMPI is however hogging available processing time on cores that are idle, it is actually making the program
7. Analysis

Figure 7.3: Sample idle times of the untuned learner on the HEPMASS dataset. The high level (1) represents a node actively working. The low level (0) represents an idle of more than 100ms. Shaded regions have frequent switches between non-idle and idle behavior of more than 100ms. The time to solution is higher due to computation time spent measuring. Across all nodes, a total of 52% of available computing time is spent idling.
Figure 7.4.: The old learner on a sample chessboard dataset. The learner is configured so that only 1 refinement cycle is run. However, just this one cycle already accounts for 60% of the available processing time, leaving only 36% for the actual training. Meanwhile, the Cholesky modification for this refinement cycle takes negligible effort. Measurements were performed with the Callgrind tool contained in Valgrind [18]. This scenario is a bad case example for the new learner, Amdahl’s Law suggests that no matter how many parallelization nodes are used, the time to solution can not be reduced past the 60% of sequential computations on the critical path.
Figure 7.5.: The new learner on the same dataset. Even though the conditions were not ideal for the parallel learner, the time to solution was reduced by 14%. Ideally, the work done by the old learner (36%) would be split into two worker portions of 18% each. This would have caused an 18% expected reduction of the time to solution. However, this the actual learner performs 4% slower than the ideal in this scenario. The idle time on the master originates from waiting for results after assigning all batches before determining the accuracy. The waiting on the workers originates from the pre-assigned working queue emptying before the master finishes refinement.
Figure 7.6.: The new learner on the same dataset without refinement. When there is no refinement that uses computation time on the master, the workers can be used to almost their full potential, spending almost the entire available computation time on learning. Using the measured load on the master for 2 workers as an indicator, the master should be able to handle at least 180 more workers for this configuration without causing slowdown. If the batch size is increased, the required computation time per worker on the master drops further, enabling the use of even higher node counts should they be desired.
run slower.

**Increasing queue sizes**  One option to reduce idle times is to increase the queue length on the master node. The queue length governs how far ahead of the workers the master is allowed to work (assigning batches and doing refinement). If the queue length is set higher, the refinement cycles will start significantly earlier as there is less waiting for results to return. This has a slightly detrimental effect to the classification accuracy, as the refinement has fewer data to work with. However, when the queue length is set very low, as is the case here, excessive idling starts to occur. In the example from Figure 7.3 this is especially evident in the large gaps between seconds 5-10 and 20-28, where the workers have finished their work queue but the master cannot assign new work as it is busy with refinement. Increasing the queue size allows the master to avoid this by assigning more jobs to workers before starting a refinement cycle. The queue size is directly influenced by the scheduler.

**Adjusting refinement**  Other options that influence the level of idling are the refinement parameters. From Figure 7.3, observe that the master is busy with refinement operations for almost 10 seconds per refinement, causing the previously mentioned idling. If the refinement frequency is doubled but the intensity halved, a similar final accuracy can be achieved while reducing the duration of each call to the refinement functor. This helps keep workers busy and is another method of raising performance.

### 7.2.2. Communication

Another aspect that causes longer run times is the communication of large data structures.

**Refinement results**  For every added grid point during the refinement, the entire level-index vector for the grid point needs to be broadcasted to all workers. For high dimensional datasets, this can cause significant overhead, especially if a lot of segmentation needs to happen.

**The Cholesky decomposition**  The transfer of the Cholesky decomposition is prone to extended runtime, as the size of the matrix is $\mathcal{O}(n^2)$, where $n$ is the number of grid points. For increasingly large grids, this can generate a lot of overhead every time the system matrix is updated, which corresponds to the refinement frequency.

**MPI Packet Size**  To combat these two issues, the size of an MPI Packet can be adjusted when compiling. The ideal packet size causes only little segmentation to happen without requiring the allocation of a lot of memory, as this memory needs to be allocated often and on many nodes. This ideal is therefore dependent on the data dimensionality and the grid size. The ideal also depends on avoiding further segmentation by MPI itself or the underlying communication protocols such as the Transmission Control Protocol.
7.3. Calibration of refinement to batches

Since the master is currently not capable of performing any learning process itself, in order to produce optimal performance the refinement monitor needs to be adjusted carefully. When the master is busy refining, all workers should have enough assigned data to learn from so that it does not run out during the period where no new batches can be assigned. When this is not the case, refinement quickly becomes a bottleneck, as seen in Figure 7.7.

Figure 7.7.: Non optimal Performance due to refinement. Refinement becomes the bottleneck as more nodes no longer cause any improvement in the time to solution. Since workers are running out of available training data while the master refines, they are forced to idle while waiting for new results to come in.

(TCP). If the learner seems to spend a lot of time communicating, adjusting the packet size can be a potential improvement.

Adjusting the packet size proved to be one of the main points in improving the runtime presented by Figure 7.3. Due to the somewhat high (28) dimensional nature of the HEPMASS dataset and the very low packet size, only 2 grid point coordinates could fit into a packet. This caused more than 30,000 packets to be generated for every refinement if the Cholesky decomposition was included. Increasing the limit caused this number to reduce drastically.
7. Analysis

7.4. Scalability Testing

7.4.1. Conducting Tests

To test different performance aspects of the program, several scaling tests will be applied. Strong scaling tests measure the time to solution for a fixed problem size on a varying number of processors. Weak scaling tests measure the time to solution for a fixed problem size per processing node on a varying number of processors. For the strong scaling test, a learning only (simple) test as well as a test with refinement (complex) will be conducted. For the weak scaling test, only a learning test will be conducted.

7.4.2. Ideal Performance

Latency  

The latency $L$ of a task is measured using the time to solution $T$, as well as the execution workload $W$, such that

$$ L = \frac{T}{W}. \quad (7.1) $$

In strong scaling tests, the total workload $W$ is kept constant, such that the latency $L$ is directly proportional to the time to solution $T$.

Speedup  

Using the latency the speedup between two architectures can be calculated as the ratio between latencies,

$$ S = \frac{L_1}{L_2}. \quad (7.2) $$

For strong scaling tests, where the workload is constant, this is equivalent to the ratio between the different times to solution,

$$ S = \frac{T_1}{T_2}. \quad (7.3) $$

Linear Speedup  

With linear speedup, the speedup by using $n$ nodes to solve a problem of fixed size should result in $S = n$. This is what will be referred to as the ideal performance in the following sections. Many algorithms typically experience sub-linear speedup, however super-linear speedup is also possible. This can, for example, be observed when parallelization improves the cache hit-rate, making the entire program run faster.

Amdahl’s Law  

For situations where not the entire program has been parallelized, Amdahl’s Law [19] can be used to predict the expected speedup of the system when the workload is fixed,

$$ S = \frac{1}{(1 - p) + \frac{p}{s}}, \quad (7.4) $$

where $s$ is the speedup of the parallelized section of the program and $p$ the proportion of the execution time spent in parallel computation. This prediction requires an estimate of the proportion $p$, which can be hard to guess or measure.
7.4.3. Simple strong scaling test

Aim To examine the effect of the number of available processing nodes on the time to solution with a fixed problem size.

Hypothesis In theory, a 2 node process should perform slightly worse than the original learner because there is only one working node, causing a slight slow down through communication overhead. Each additional working node is expected to reduce the time to solution, with a decreasing turnover for a high number of nodes.

Methodology The tests will be conducted with the original learner, as well as the new learner with world sizes from 2 through 8, where 8 nodes exhaust all available cores on the system. For these tests, the chessboard data set with 2 dimensions was chosen because the low dimensionality improves data set loading times (decreasing the total time required to conduct measurements) and the data set can be scaled to an arbitrary number of data points as it is artificial. The number of training data points was fixed at 15,000,000. This size was set as a compromise between the requirement to have a sufficiently long runtime on 8 threads to prevent jitter through interference by other processes and the processing time required to load large datasets into each node’s memory. For the simple test no refinement occurs and the batch size is constant at 300,000 data points. Note that, again, OpenMP was purposefully disabled for these tests to avoid cross-thread interference. Each learner was trained and tested 20 times, constituting a runtime of over 6 hours for the entire experiment.

Results See Figure 7.8 for a summary of the results. Surprisingly, the 2 node (1 master, 1 worker) configuration actually improved the time to solution slightly. This is probably due to some overhead from the original learner being transferred to the master, allowing the worker to spend all the available computation time solely on learning. This causes a slight parallelization improvement instead of none at all. Furthermore, it may be observed that there is less variance in the time to solution on the new learner in comparison to the old learner. This indicates more reliable performance on part of the new learner.

Additionally, the overhead in relation to the ideal performance was measured in Figure 7.9. The ideal performance was set according to linear scaling of the original learner. The strong scaling efficiency can be calculated using

\[ \eta_{\text{strong}} = \frac{t_1}{n * t_n}, \]

(7.5)

where \( t_1 \) is the time to solution for 1 node computation and \( t_n \) the time to solution for \( n \) node computations. The resulting coefficient indicates efficiency as a percentage of ideal linear scaling, as seen in Figure 7.10. It may be observed that for almost all tests, save for outliers with a 2 node and 8 node process, the efficiency is consistently higher than 90%.
7. Analysis

![Time to Solution on Strong Scaling Test](image)

Figure 7.8.: Time to solution for number of nodes in the simple strong scaling test. As expected, the average time to solution loosely correlates to the formula for linear speedup $t_0 \times \frac{1}{n}$. A slight deviation from this trend is observed for a 2 node setup.

**Evaluation** From the overhead results, the measurements for 8 nodes perform slightly worse than the measurements for 7 cores. This is likely due to all the host processing cores being used for computation, causing operating system and other tasks to interrupt learning as there are no other free cores to work on. Thus, the performance result on 8 cores is likely less representative as noise through incontrollable variables is increased.

**Conclusion** Overall, the strong scaling test suggests that performance is very close to ideal linear scaling when no refinement takes place.

7.4.4. Complex strong scaling test

Having shown that isolated learning performs well, a more complex test with intermediate refinements will now be performed to emulate slightly more realistic conditions.

**Aim** To examine the effect of refinement and the number of available processing nodes on the time to solution with a fixed problem size.
Figure 7.9.: Time to solution for number of nodes in the simple strong scaling test. The overhead was measured in relation to the ideal linear performance scaling of $t_0 \times \frac{1}{n}$, where the benchmark of the original learner was used as factor $t_0$. The high overhead on the two node setup comes from only one worker actually training. The increase in overhead on the eight node setup comes from interference by other processes as the processor capacity is reached.

Figure 7.10.: Efficiency in relation to ideal linear scaling
7. Analysis

**Hypothesis**  The total time spent on refinement should limit the possibility to decrease the time to solution by increasing the number of available nodes. The time spent on refinement is directly controlled by the refinement frequency and the number of points per refinement cycle. An increase in all of these variables should result in a decrease of parallel performance.

**Methodology**  The same setup as in the last tests will also be used for these tests. To examine the effect of the refinement frequency, the parallelization will be tested for 7 different refinement frequencies. One of these will be with no refinement at all, for comparison. Three tests will be with 1 refinement cycles at different positions in the learning phase. Three more will use higher refinement frequencies. There is also an additional parameter specifying the maximum number of refinements. However, since this parameter is semantically almost identical to controlling the refinement frequency, only the test with adjusting the refinement frequency will be conducted.

**Results**  See Figure 7.11 for a summary of the results. As expected, the refinement frequency can severely limit the ability of the learner to decrease the time to solution by parallelizing. For one refinement cycle, after the addition of one thread (master does refinement, worker does learning) the time to solution is halved. This suggests that there is approximately equal time spent learning and refining on this setup. The time to solution could still be improved slightly with more nodes in some cases, however the refinement causes bottlenecking on the critical path. This restrains the effectivity of adding additional learning nodes. As the frequency of refinements increases, the learners ability to reduce the time to solution is further diminished as the sequential refinement takes up a larger percentage of the overall processing time.

**Evaluation**  The refinement frequency dictates how often a refinement cycle and system matrix update is necessary. The number of refined points per refinement cycle directly affects the size of the grid modifications transferred and the new size of the system matrix decomposition, which corresponds to the number of grid points squared.

**Conclusion**  Refinement needs to be adjusted carefully to avoid tampering the learner efficiency. As soon as the time to solution converges against the total time required for all refinement steps, the efficiency of the learner drops sharply. If the time to solution needs to be reduced further, one is forced to lower either the refinement frequency or the refinement intensity to reduce the time required in refining. Unfortunately, both of these adjustments will likely affect the resulting accuracy on the learner.

7.4.5. Weak scaling test

Weak scaling is a type of test where the problem size is kept identical per node, and the time to solution is measured for varying number of nodes. In an ideal scenario for an algorithm of linear complexity, the weak scaling of an algorithm should not affect the
7.4. Scalability Testing

Figure 7.11.: Time to solution for number of nodes and different refinement frequencies in the complex scaling test. 0 refinement cycles shows the scaling in a best case scenario. 1 refinement cycle is already sufficient to prevent a speedup of more than 2.5. It may be observed that for earlier refinement, the ability to scale is hampered slightly less. Every additional refinement cycle increases the time to solution and decreases the potential speedup.

time to solution for an increasing number of nodes. If this is not the case, then it is possible that the overhead through parallelization increases with additional nodes, or the algorithm is not truly in $O(n)$ [20].

Aim To examine the effect of the number of available processing nodes on the time to solution with a fixed problem size per node.

Hypothesis Ideally, the number of available processing nodes should not affect the time to solution in a learning only experiment because the total problem size grows with the number of nodes.

Methodology The same setup as in the last tests will also be used for these tests. This test will be conducted twice. Once with the already previously utilized 1-8 node configuration, and once with the 1-32 node configuration.
7. Analysis

Results See Figure 7.12 for a summary of the results for the 1-8 node configuration. The overhead increase for 2 nodes already observed in previous tests can also be observed here. The higher node counts converge towards a value that exceeds the performance of the original learner. Figure 7.13 shows the same experiment run on a different system with up to 32 nodes. Interestingly, the second test shows an anomaly at 4 nodes, the performance of which would have been expected to be significantly better than for 2 nodes. The test shows good scaling up to the point where the full load of the cluster is reached. From these results, it is now possible to calculate the weak scaling efficiency according to the following formula:

\[ \eta_{\text{weak}} = \frac{t_1}{t_n}. \]  

(7.6)

These median efficiencies can be seen on Figure 7.14. They show that not only was the parallelization efficient, they actually scale with better than linear efficiency. This is likely due to performance optimizations that took place during the parallelization that are not present in the original learner itself.

Conclusion As already observed previously, the communication overhead is large for a very low number of nodes as the ratio between master and workers is low, causing a lower weak scaling efficiency. This is evident from the results of test 1. Fortunately, the time to solution quickly converges to a fixed constant, suggesting that the overhead does not correspond to the number of nodes for the node counts measured. To ensure this also applies for higher node counts, test 2 shows the results when run on up to 32 nodes. Both tests show better than linear scaling for certain conditions.

Evaluation Each test shows increasing uncertainty when approaching the full load of the cluster, similar to problems in previous tests. These experiments may be improved by ensuring that the measuring environment is kept free from interference by other processes. This could be done by reserving resources for other use while measuring to ensure the operating system has enough computation time without having to subtract from the time spent learning.
7.4. Scalability Testing

Figure 7.12.: Time to solution for number of nodes in the simple weak scaling test. Similar to the additional overhead in the strong scaling test, the efficiency for the weak scaling test is not ideal for a 2 node setup, however it improves with every additional node added. As seen in the previous strong scaling tests, the higher variance on the 8 node setup originates from interference by other processes run by the operating system. The weak scaling test shows that growing the problem size to fit the number of nodes used does not adversely affect the time to solution for high node numbers.
Figure 7.13.: Time to solution for number of nodes in the simple weak scaling test run on a setup with up to 32 nodes. The same performance dip as in Figure 7.12 near the low node counts of 2 and 4 may be observed. Interestingly, 4 nodes perform slightly worse than 2 nodes, where in theory they should perform significantly better. For node counts 16, 24, and 30 a performance better than the original learner can be observed. Another slight performance dip is noticeable as the cluster approaches its full load with 32 nodes.
Figure 7.14: Time to solution for number of nodes in the simple weak scaling test. The values higher than 100% efficiency imply that not only has the solution been parallelized efficiently in comparison to the original learner, it also shows that several routines have been optimized as compared to the original learner during the parallelization process. A similar drop in efficiency already observed in the previous tests can be seen for low node counts.
8. Further improvements

The current implementation of the parallel learner already signals a significant improvement over the originally sequential learner. However, the implementation is still in an early stage of development with the goal of fully parallelizing all important aspects of training the learner. Several open points that represent opportunity for decreasing the time to solution, or the increasing the overall efficiency or robustness of the learner are outlined below.

8.1. Streaming data sets

Currently, both the old learner and the new learner hold the entirety of their data sets in memory after pre-loading them from storage. This mechanism is especially wasteful as it occurs on every node, causing nodes to load the entire dataset, most of which they will not end up using. This causes problems when data sets are large enough to not fit into memory, especially during the scaling tests were the node count was high in relation to the available memory. Potential solutions include sending the actual data to train on along with the batch assignment request, therefore only retaining the entire set on the master node, or implementing a mechanism that allows partial loading of datasets from their storage. The latter solution is expected to perform better on computing setups that store data in network attached storage, as the latter does not load data into the master only for it to be redistributed further. Both will significantly lower memory usage for high node counts.

8.2. Improve System Matrix Updates

Currently, it is necessary to transfer the entire system matrix for every refinement cycle. For the Cholesky decomposition, this currently constitutes a transfer of $O(n^2)$ elements. A potential for improvement has already been identified; the system matrix is a lower triangular matrix, requiring the transfer of only half the values. However, in future it is possible to attempt to detect changed values in the decomposition and transfer only the delta values. Whether this will facilitate further performance improvements remains to be inspected.

Transfer improvement Currently, the system matrix is first transferred from the worker that originally updated it to the master, thereafter rebroadcasting it to all workers from the master. This stems from the requirement that a broadcast needs to know its
8. Further improvements

root before it is started on any node. This causes an overhead that is slightly unnecessary. Instead, it would be possible to build a system where there are \( n \) open broadcasts at any time, one for each possible root node. Note that MPI broadcasts cannot be canceled, they would need to be completed with empty data at shutdown.

8.3. Parallel-Refinement

The parallelization of the refinement functors is a high priority for further decreasing the time to solution by increasing the node count. As previously shown in Figure 7.7, parallelization can easily represent a bottleneck, especially for small-medium sized datasets where the refinement frequency is high. While the effect can be mitigated somewhat, this only delays the loss of efficiency to higher node counts rather than removing it entirely. However, since there are multiple different implementations for refinement functors available, this probably requires re-implementing every functor to support parallel computing architecture.
9. Conclusion

Through analysis of the current implementation of the classification learning algorithm, a new parallel learning algorithm was planned and implemented. The design included analysis of different parallelization approaches and creating a communication plan for the exchange of data. The implementation realized the communication plan as well as several support structures that facilitate the export, communication, and re-import of data on the different nodes.

Tests were conducted to show that the new algorithm reduces the resulting accuracy of the classifier on the validation set by less than 0.2% while decreasing the time to solution significantly. Furthermore, both strong and weak scaling tests have been conducted to calculate the average efficiency of the parallel implementation.

As a result, a close to ideal performance could be demonstrated for higher node counts. When using 5 nodes to solve the training phase in parallel already more than 79% of the time to solution could be eliminated. In some tests, when compared to the sequential implementation of the algorithm, a better-than-linear scaling could be observed.

A potential for bottle-necking has been identified in the refinement of the sparse grid, a currently sequential and time-intensive task that has not been parallelized as of today. However, a potential to minimize the effect of refinement on the time to solution has been shown to avoid this bottle-neck as far as possible. While the current solution already goes a long way to reducing the time to solution, further tasks to improve the efficiency have been identified.

Due to the work delivered in this thesis, it is now possible to train a sparse grid classifier on data sets that were previously too large to handle in a time effective manner. This is expected to strengthen the level of competitiveness for the SG++ library in solving classification problems quickly and accurately.
Appendices
A. Experiment Results

<table>
<thead>
<tr>
<th>Accuracy (%)</th>
<th>Original</th>
<th>2 thread</th>
<th>4 thread</th>
<th>6 thread</th>
<th>8 Thread</th>
<th>12 Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>89.8</td>
<td>89.5</td>
<td>89.7</td>
<td>89.6</td>
<td>89.7</td>
<td>90.1</td>
</tr>
<tr>
<td>Min</td>
<td>89.0</td>
<td>88.8</td>
<td>89.1</td>
<td>89.1</td>
<td>89.1</td>
<td>89.8</td>
</tr>
<tr>
<td>LQ</td>
<td>89.6</td>
<td>89.3</td>
<td>89.6</td>
<td>89.5</td>
<td>89.4</td>
<td>90.1</td>
</tr>
<tr>
<td>Median</td>
<td>89.7</td>
<td>89.5</td>
<td>89.7</td>
<td>89.6</td>
<td>89.6</td>
<td>90.1</td>
</tr>
<tr>
<td>UQ</td>
<td>90.0</td>
<td>89.7</td>
<td>89.8</td>
<td>89.7</td>
<td>89.9</td>
<td>90.1</td>
</tr>
<tr>
<td>Max</td>
<td>90.5</td>
<td>90.1</td>
<td>90.2</td>
<td>90.0</td>
<td>90.4</td>
<td>90.2</td>
</tr>
</tbody>
</table>

Table A.1.: Statistics for the Accuracy Test (50 trials) from Figure 7.1.

<table>
<thead>
<tr>
<th>Timing (s)</th>
<th>Old learner</th>
<th>3 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>46.30</td>
<td>54.91</td>
</tr>
<tr>
<td>Min</td>
<td>45.56</td>
<td>54.34</td>
</tr>
<tr>
<td>LQ</td>
<td>45.77</td>
<td>54.59</td>
</tr>
<tr>
<td>Median</td>
<td>45.79</td>
<td>54.63</td>
</tr>
<tr>
<td>UQ</td>
<td>46.94</td>
<td>55.45</td>
</tr>
<tr>
<td>Max</td>
<td>47.41</td>
<td>55.52</td>
</tr>
</tbody>
</table>

Table A.2.: Statistics for the Bad Performance Test (5 trials) from Figure 7.2.

<table>
<thead>
<tr>
<th>Timing (s)</th>
<th>Original</th>
<th>2 node</th>
<th>3 node</th>
<th>4 node</th>
<th>5 node</th>
<th>6 node</th>
<th>7 node</th>
<th>8 node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>66.20</td>
<td>60.29</td>
<td>29.35</td>
<td>19.90</td>
<td>16.90</td>
<td>13.56</td>
<td>11.75</td>
<td>11.63</td>
</tr>
<tr>
<td>LQ</td>
<td>74.19</td>
<td>60.84</td>
<td>30.57</td>
<td>20.87</td>
<td>17.37</td>
<td>14.42</td>
<td>12.36</td>
<td>12.26</td>
</tr>
<tr>
<td>Median</td>
<td>84.3s</td>
<td>61.5s</td>
<td>31.2s</td>
<td>21.6s</td>
<td>17.6s</td>
<td>14.8s</td>
<td>12.9s</td>
<td>12.5s</td>
</tr>
<tr>
<td>UQ</td>
<td>86.4s</td>
<td>62.96</td>
<td>31.78</td>
<td>22.42</td>
<td>18.33</td>
<td>14.94</td>
<td>13.46</td>
<td>12.70</td>
</tr>
<tr>
<td>Max</td>
<td>89.42</td>
<td>65.63</td>
<td>32.79</td>
<td>23.56</td>
<td>19.26</td>
<td>15.29</td>
<td>13.84</td>
<td>13.93</td>
</tr>
</tbody>
</table>

Table A.3.: Statistics for the Strong Scaling Test (20 trials) from Figure 7.8.
### A. Experiment Results

<table>
<thead>
<tr>
<th>Timing (s)</th>
<th>Original</th>
<th>2 node</th>
<th>3 node</th>
<th>4 node</th>
<th>5 node</th>
<th>6 node</th>
<th>7 node</th>
<th>8 node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>4.88</td>
<td>7.65</td>
<td>5.33</td>
<td>4.41</td>
<td>4.17</td>
<td>4.13</td>
<td>4.08</td>
<td>4.17</td>
</tr>
<tr>
<td>LQ</td>
<td>4.49</td>
<td>6.47</td>
<td>5.00</td>
<td>4.29</td>
<td>4.14</td>
<td>4.08</td>
<td>3.99</td>
<td>4.03</td>
</tr>
<tr>
<td>Median</td>
<td>4.82</td>
<td>7.04</td>
<td>5.16</td>
<td>4.35</td>
<td>4.17</td>
<td>4.13</td>
<td>4.10</td>
<td>4.12</td>
</tr>
<tr>
<td>UQ</td>
<td>5.14</td>
<td>8.29</td>
<td>5.59</td>
<td>4.41</td>
<td>4.20</td>
<td>4.21</td>
<td>4.13</td>
<td>4.21</td>
</tr>
<tr>
<td>Max</td>
<td>5.69</td>
<td>10.07</td>
<td>6.64</td>
<td>4.94</td>
<td>4.32</td>
<td>4.26</td>
<td>4.22</td>
<td>4.96</td>
</tr>
</tbody>
</table>

Table A.4.: Statistics for the Weak Scaling Test (20 trials) from Figure 7.12.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>30</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>0.60</td>
<td>1.46</td>
<td>1.36</td>
<td>1.23</td>
<td>1.45</td>
<td>1.01</td>
<td>1.05</td>
<td>1.25</td>
<td>1.62</td>
<td>1.03</td>
</tr>
<tr>
<td>AVG</td>
<td>0.83</td>
<td>1.06</td>
<td>1.08</td>
<td>0.92</td>
<td>0.84</td>
<td>0.78</td>
<td>0.79</td>
<td>0.78</td>
<td>0.89</td>
<td>0.85</td>
</tr>
<tr>
<td>MIN</td>
<td>0.25</td>
<td>0.63</td>
<td>0.62</td>
<td>0.71</td>
<td>0.40</td>
<td>0.54</td>
<td>0.56</td>
<td>0.59</td>
<td>0.63</td>
<td>0.62</td>
</tr>
</tbody>
</table>

Table A.5.: Statistics for the Weak Scaling Test with high node counts (6 trials) from Figure 7.13.

<table>
<thead>
<tr>
<th>Type of activity</th>
<th>Estimated CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train specific</td>
<td>7412922461</td>
</tr>
<tr>
<td>Refine</td>
<td>12425109660</td>
</tr>
<tr>
<td>Get Accuracy</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky Modification</td>
<td>599540</td>
</tr>
<tr>
<td>Other</td>
<td>812955737</td>
</tr>
</tbody>
</table>

Table A.6.: Callgrind Measurements for the Old Learner with Refinement. Time to solution was 480s during instrumentation. From Figure 7.4.

<table>
<thead>
<tr>
<th>Type of activity</th>
<th>Estimated Cycles (Master)</th>
<th>Worker 1</th>
<th>Worker 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refine</td>
<td>3184757172</td>
<td>10635874212</td>
<td>10495125489</td>
</tr>
<tr>
<td>Alpha Merge</td>
<td>3193070</td>
<td>3799721</td>
<td>9824090</td>
</tr>
<tr>
<td>Wait</td>
<td>3391304468</td>
<td>1726341278</td>
<td>1737863621</td>
</tr>
<tr>
<td>Cholesky Update</td>
<td></td>
<td>1445374</td>
<td>1445374</td>
</tr>
<tr>
<td>Work Batch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.7.: Callgrind Measurements for the New Learner with Refinement from Figure 7.5.
<table>
<thead>
<tr>
<th>Type of activity</th>
<th>Estimated Cycles (Master)</th>
<th>Worker 1</th>
<th>Worker 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refine</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha Merge</td>
<td>21628</td>
<td></td>
<td>6715139</td>
</tr>
<tr>
<td>Wait</td>
<td>5517702834</td>
<td>6786080</td>
<td></td>
</tr>
<tr>
<td>Cholesky Update</td>
<td></td>
<td>3799721</td>
<td>9824090</td>
</tr>
<tr>
<td>Work Batch</td>
<td></td>
<td>1684638725</td>
<td>1671327266</td>
</tr>
<tr>
<td>Other</td>
<td>578970</td>
<td>3698000</td>
<td>3697003</td>
</tr>
</tbody>
</table>

Table A.8.: Callgrind Measurements for the New Learner without Refinement from Figure 7.6.
Bibliography


[14] MPICH, Web pages for mpi and mpe,


[17] The C++ Resources Network, C++ reference,

