Design, implementation, and optimization of an advanced I/O Framework for Parallel Support Vector Machines

Sigurður Páll Behrend

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
University of Iceland
2018

DESIGN, IMPLEMENTATION, AND OPTIMIZATION OF AN ADVANCED I/O FRAMEWORK FOR PARALLEL SUPPORT VECTOR MACHINES

Sigurður Páll Behrend

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Computer science

M.Sc. committee
Prof.Dr.-Ing. Morris Riedel
Dr. Helmut Neukirchen

External examiner
Dr. Lars Hoffmann

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, June 2018

Design, implementation, and optimization of an advanced I/O Framework for Parallel Support Vector Machines
60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Computer science

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Tæknigarður - Dunhaga 5
107, Reykjavik, Reykjavik
Iceland

Telephone: 525 4000

Bibliographic information:
Sigurður Páll Behrend, 2018, Design, implementation, and optimization of an advanced I/O Framework for Parallel Support Vector Machines, M.Sc. thesis, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland.

Reykjavik, Iceland, June 2018

I made dis

Abstract

The thesis goal is to improve the I/O performance of the PiSVM suite of parallel and scalable tools used for machine learning on HPC platforms. This is achieved by analyzing the current state of I/O and then designing an I/O framework that enables PiSVM programs to read and write data in parallel, using the HDF5 library. HDF5 is a highly scalable library that is used in HPC but its use is not widespread. The thesis implements the design into the PiSVM toolset as a proof-of-concept. A parser will be added to the PiSVM suite that converts data from the currently used SVMLight format into HDF5 format. A 3.45% overall reduction in execution time was achieved in PiSVM-Train on the Indian Pines Raw dataset. A 4.88% overall reduction in execution time was achieved in PiSVM-Predict on the Indian Pines Raw dataset. Read and write times were improved by a bigger percentage, upward to 98% reduction in read and write times. This can be attributed to the design of the I/O framework and usage of advanced data storage features that HDF5 offers. A further significant result is a reduction of data file size by 72% and a reduction of model file size by 24% on the Indian Pines Raw dataset. In practice, any work with PiSVM will gain significant benefits from the work done in this thesis. Whole research groups tend to have multiple copies of the data, working with different feature engineering techniques and As PiSVM is used in many supercomputing centers and by multiple research groups, the gains are significant.

Útdráttur

Tilgangur ritgerðarinnar er að bæta les og skrifhraða í PiSVM tólasafninu sem notað er í vélnámi og stórvirkri tölvuvinnslu. Þetta verður gert með því að skoða núverandi les og skrifaðgerðir, og hanna viðbætur sem gera PiSVM tólum kleift að lesa og skrifa gögn hliðrænt á HDF5 sniði. HDF5 er stigfrjálst og sveigjanlegt skráarform sem er notað í stórvirkri tölvuvinnslu. Lausnin verður útfærð í PiSVM tólasafninu til að sannreyna virkni. Við klasasafnið verður bætt við flokkara sem getur lesið gögn á SVMLight textasniði og skrifar þau út í skrá á HDF5 bitasniði. Það tókst að lækka heildar keyrslutíma PiSVM-Train um 3.45% á Indian Pines Raw gagnasafninu. Það tókst að lækka heildar keyrslutíma PiSVM-Predict um 4.88% á Indian Pines Raw gagnasafninu. Mikið meiri lækkun á keyrslutíma les og skrifaðgerða náðist, allt upp í 98% lækkun. Þessi aukning á keyrsluhraða er vegna hönnunar skráarsniðs og eiginleika sem HDF5 klasasafnið býður upp á við geymslu á gögnum á HDF5 skráasniði. Aukalega náðist fram smækkun á stærð gagnaskráa um 72% og smækkun model skráa um 24% á Indian Pines Raw gagnasafninu. Þar sem verið er að vinna með PiSVM á mörgum ofurtölvum í heiminum og mörg eintök af gögnum eru iðulega geymd á meðan verið er að vinna með þau, þá er þetta töluverður kostur.

Table of contents

List of Figures

List of Tables

Listings

Abbreviations

- ANN Artificial Neural Network
- AVIRIS Airborne visible/infrared imaging spectrometer
- CPU Central Processing Unit
- GPU Graphics Processing Unit
- HDF Hierarchical Data Format
- HDF5 Hierarchical Data Format version 5
- HPC High Performance Computing
- HTC High Throughput Computing
- I/O Input/Output
- JSC Jülich Supercomputing Centre
- JURECA Jülich Research on Exascale Cluster Architectures
- ML Machine Learning
- MPI Message passing interface
- MLlib Apache Spark's scalable machine learning library
- PVFS Parallel Virtual File System
- RBF Radial Basis Function
- RS Remote Sensing
- SLURM Simple Linux Utility for Resource Management
- SVM Support Vector Machine
- *NIX Used to refer to operating systems that are like Unix

Acknowledgements

I would like to thank my thesis advisor Prof.Dr.-Ing. Morris Riedel of the Forschungszentrum Jülich (Jülich Supercomputing Centre) and University of Iceland, for his advice, help, and support in this project. He gave me almost full control over all aspects of the project and offered insight and advice whenever I needed it.

Huge thanks must go to Forschungszentrum Jülich for allowing me to work on their cutting edge HPC systems. I sincerely hope that my little addition to their tools will serve them well.

I must express my very profound gratitude to my friend Kveldúlfur Þrastarson for all his help during our Masters education. His help, support and advice throughout the years have been invaluable.

My family deserves special thanks for always supporting me in my pursuit of knowledge and education.

Finally I want to thank my girlfriend Bergný Sjöfn Sigurbjörnsdóttir, for everything.

1 Introduction

1.1 Motivation

Remote sensing, the process of data acquisition without making physical contact with the object, can create very large, detailed and complex datasets. Those datasets need to be stored and processed before they can be used. This requires either High-Performance-Computing (HPC) or High-Throughput-Computing (HTC) and large amount of storage space. Because of slow, serial I/O methods of reading and writing, these large datasets can be very time consuming to process into a format that can then be used to learn anything from. The need for some better I/O solution is becoming more and more apparent as the size of datasets continue to grow at a very fast rate.

This thesis is a groundbreaking first step in developing support for a newer, powerful and flexible parallel I/O framework for Parallel Support Vector Machines. An already popular suite of HPC software will be used to evaluate the performance of the I/O framework once it is complete. This suite is the parallel and scalable implementation of SVMs called PiSVM. It will receive full integration of the parallel I/O framework and will be used to run tests on real-world datasets in order to evaluate performance improvements.

The design and implementation will be done in such a way that makes it easy to implement it in other applications. This requirement will be evaluated in another M.Sc. thesis by Kveldúlfur Þrastarson[1] as he will be using the parallel I/O framework implementation in his CascadeSVM thesis work.

For the purposes of this thesis, it will be necessary to study the inner workings of PiSVM and especially the I/O strategy it uses. It will then need to be determined where processing can be improved by implementing parallel I/O instead of the serial I/O that the current implementation uses. It will also be necessary to study the inner workings of the libraries available. To look at what options are available for scalable storage, what features libraries offer that can benefit the I/O framework, and if they are flexible enough to use in order to fulfill the requirements and answer fully the research questions posed in the thesis. It will be necessary to design file layouts

that best fit the requirements imposed by the driving remote sensing application in general and the used datasets in particular.

Test methods will have to be devised that accurately measure the changes in I/O performance. These tests must also confirm that the processing part of the PiSVM suite has not been altered, neither in terms of processing speed or data evaluation results, the accuracy of evaluation must not change between implementations.

1.2 Research questions

1. How can I/O speed best be increased in the PiSVM suite of HPC tools?

2. Currently there exist several different libraries that could potentially increase I/O speeds in the PiSVM suite, which of them offers the best chance of meeting the primary goal of speed improvement?

3. What work will be required to integrate support for the chosen framework into PiSVM-Train, PiSVM-Predict and PiSVM-Scale?

4. How should the chosen framework be evaluated after integration?

5. Some of the available I/O libraries offer additional functionality, like compression and chunking. What additional functionality can be implemented to achieve the primary thesis goal of speed improvement, and what functionality can be used to achieve secondary goals of reduced file size without sacrificing read and write speed?

1.3 Structure of Thesis

Chapter 1 discusses the motivation behind this thesis and what are the main research questions that this thesis seeks to answer.

Chapter 2 will lay down the foundation on which the thesis bases other chapters. It will give a broad overview of the relevant concepts so that the reader will have a better understanding of later chapters where topics specific to the thesis will be discussed in more detail.

Chapter 3 exposes the reader to work currently being done that is related to this thesis and seeks to give a broad explanation of each of those topics without going into too much detail.

Chapter 4 seeks to explain the architectural design and implementation that was devised as part of the thesis work. It seeks to explain how the PiSVM tools worked prior to the improvements made on them, and what modifications were made in order to integrate the parallel I/O framework.

Chapter 5 gives an overview of the evaluation environment, as well as going into detail about the tests used to evaluate the parallel I/O framework and its performance. It sets forth the results of these tests and discusses each one relative to the research questions posed in chapter 1, and the architectural designs from chapter 4.

Chapter 6 draws up a conclusion for the thesis and discusses what research questions were adequately answered. It also gives hints for the application of this parallel I/O framework and goes over ideas of future work that can be based on this thesis.

2 Foundations

2.1 Remote sensing

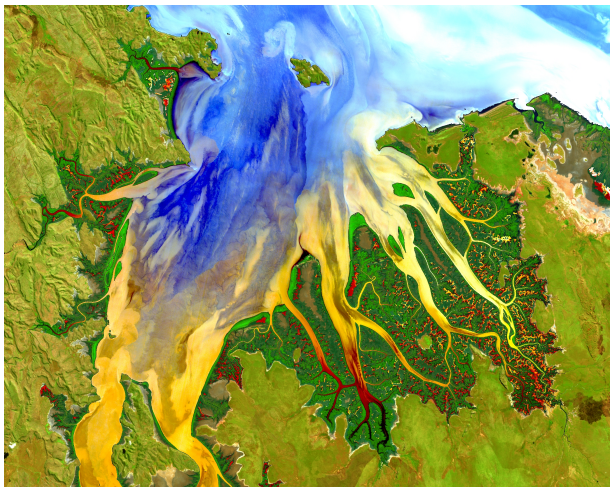

Figure 2.1: Remote sensing satellite image of western Australia, taken with the LANDSAT 8 Satellite Sensor (15m)[1].

Remote sensing is the process of acquiring data about an object remotely, without making physical contact with it or interacting with it. Remote sensing often refers to the use of satellite and aircraft-based sensor technologies to detect and classify objects on Earth, but it can also be used on objects on other planets or other entities in space. The sensor technologies are varied, they can be based on propagated signals like electromagnetic radiation, either in the visible spectrum of light or outside of it. The sensor technology can be split into active and passive remote sensing. With active remote sensing, the signal is emitted by the sensor and the reflection is observed. With passive remote sensing, the reflection of ambient signals or signals that are emitted by the object itself is recorded by a sensor[2].

[1] `https://content.satimagingcorp.com/static/galleryimages/landsat-8-satellite-image-western-australia.jpg`

Remote sensing is used in numerous fields, including geography, land surveying, hydrology and ecology, oceanography, glaciology, geology etc. An example product of Remote sensing is figure 2.1. An image like that can be compared to images taken at later dates, in order to understand how the land changes and why. It also has military, intelligence, commercial, economic, environmental, planning and humanitarian applications[3]. The particular driving thesis application is primarily focused on classifying land cover via remote sensing images[2].

The product of remote sensing is the data gathered about aspects of the object being observed (the state), as well as other variables whose source is not the object itself, but may offer information about the object and can be detected and measured (the observation). This data can very often be huge and is divided into several processing levels. NASA proposed levels 0 through 4 with the following distinction[4]:

- Level 0: Reconstructed, unprocessed data at full resolution, with artifacts (e.g. communications headers, duplicate data) removed. This level is the foundation upon which all subsequent data sets are produced.
- Level 1A: Reconstructed, unprocessed instrument data at full resolution, time-referenced, and annotated with ancillary information.
- Level 1B: Level 1A data that has been processed to sensor units (i.e., radar backscatter cross section, brightness temperature, etc.). Not all instruments will have a Level 1B equivalent.
- Level 2: Derived geophysical variables at the same resolution and location as Level 1 source data. This is the first level that is usable in most scientific applications. Thus its value is much greater than the lower levels, but also less voluminous because data has been reduced temporally, spatially or spectrally.
- Level 3: Variables mapped on uniform space-time grid scales, usually with some completeness and consistency. Datasets at this level are generally smaller than its predecessors and are often small enough to allow to combine data from different sources.
- Level 4: Model output or results from analysis of lower-level data. This is the final level, observations can be based on Level 4 data but the data is generally not refined further.

These datasets have become larger and larger with better sensor technology and increased storage capacity. Huge datasets at level 0 have to be processed into higher level datasets with the aid of larger and larger computers that utilize parallel computing and machine learning.

HPC and HTC are the two dominant computing paradigms. HPC tasks tend to need large amounts of computing power for short periods of time. HPC offers good performance with its cores and communicates effectively via a dedicated interconnection between them. HTC consist of moderately powerful hardware only interconnected

via Ethernet, and HTC tasks are rather focused on each core/node just doing localized work with little need for communication between nodes. HPC systems thus usually offer a parallel file system that enables the parallel I/O required to process the ever increasing resolutions and sizes of remote sensing data.

2.2 Parallel computing

A computation task is traditionally solved by constructing an algorithm that is executed as a serial stream of instructions, only one instruction is executed at a time, after that execution is finished, the next one is executed, until no more instructions remain, the task is then considered complete and a solution is returned[5].

Parallel computing breaks a computational task down into smaller, preferably independent, subtasks. These subtasks can then be solved near simultaneously, in multiple processes. After all subtasks are finished the results are combined and a solution is returned[5].

Optimally the speedup should be linear, doubling the number of processes should reduce the runtime by half. But some algorithms are inherently hard to parallelize. For example dividing up a task and combining the results are often linear tasks, executed by a single process. If dividing and combining are a big part of the total execution time then the speed-up will be lower than linear, at worst the speed-up becomes only marginally faster than if the task was executed in serial[5] [6].

Multiple types of parallelism exist, but most relevant to the work being presented in this thesis is Task Parallelism. A task is divided into sub-tasks and each sub-task is then assigned to specially designed hardware for execution[5].

Hardware that runs parallel computations is varied, it ranges from a single processor with multiple cores, to multiple standalone computers that are networked together to share tasks, to gigantic racks, each filled with specialized hardware, and high-speed fiber network to connect them all together[5].

Parallel computing uses a communication standard called MPI that is now widely used in High Performance Computing. In distributes memory systems MPI defines a system that allows processes to communicate with each other, as they do not have direct access to each others memory and so cannot communicate directly.

An aspect of HPC hardware is that not only can the processors be used in parallel but I/O can also be done in parallel on top of parallel file systems. This is an important consideration for data-intensive applications, like those used in machine

learning.

2.3 Machine learning

Machine learning aims at giving computers the ability to solve problems without being explicitly programmed on how to solve them. It evolved from the study of pattern recognition and computational learning theory in artificial intelligence. The aim is to construct algorithms that can learn from data and make data-driven predictions or decisions[7].

Sample data is used to build a model by training the algorithm on it. Data can be processed and parameters tweaked to achieve the highest possible accuracy. The algorithm is then tested on related data, for which the same model should also fit, and if that also achieves high accuracy then the model is good and can be put to use in classifying data in real-world scenarios. Machine learning is used in fields like computer vision, optical character recognition, and remote sensing. Parallel computing becomes more important as the size of datasets grows because it can do classification where serial tools fail at classification (e.g. because data does not fit into memory). Parallel computing also offers faster classification of data[8].

Supervised learning is the task of constructing a model from labeled training data. A classification algorithm examines the data and infers a function that can distinguish between labels. This algorithm is used to construct a classification model. The model learns on the training data and is validated on labeled test data. Once good accuracy is achieved the model can be used to classify unseen data[9].

Unsupervised learning also infers a function from the training data but the data is unlabeled. This adds the task of having to find any hidden structure within the data in order to distinguish between classes. Since the data is unlabeled it is also harder to evaluate a potential solution as there is no classification to compare results to[9].

For the sake of completeness, it is important to mention that a third learning model exists, reinforcement learning. It is concerned with teaching software agents to take actions in an environment so as to maximize some notion of cumulative reward. This third method, and unsupervised learning, are outside the scope of this thesis.

2.4 Support Vector Machines

Within the field of Machine learning, SVMs are supervised learning models with associated learning algorithms that analyze data for classification and regression analysis[10].

The process of using SVMs is threefold, training, testing and predicting. In the training stage the SVM is given a set of labeled training examples and using those examples it tries to create an optimal hyperplane which categorizes new examples. There can be many valid hyperplanes but the optimal one is situated in the middle of the two classes and as far as possible from all points (as shown in figure 2.2). This is the maximum margin hyperplane and those data points that define its outer lines are called Support Vectors (SVs) The hyperplane is then optimized by trying to find the maximum margin, the data points that are situated on the margin are called Support Vectors (SVs).

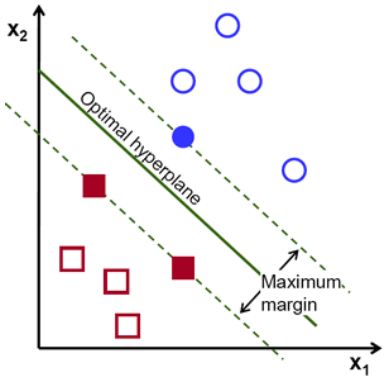

Figure 2.2: Diagram of the SVM hyperplane concept. Solid colored shapes are support vectors[2].

Figure 2.2 shows what the optimal hyperplane is and how the maximum margin is achieved. Once the maximum margin hyperplane has been found the SVs are written into a model file. Testing is then performed by using the model file to predict the classification of the test data. Since the test data is labeled it is possible to compare the prediction to the actual value and calculate the accuracy of the model. Training and testing data need to be related, so usually a whole dataset is split up into training and testing parts.

Once an accurate model has been created, it is then used as input for the prediction phase where unclassified data is classified into categories based on the model file[10].

[2] `https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html`

The original SVM algorithm was invented in 1963 but at first only supported linear classifiers. In 1992 Bernhard E. Boser suggested a way to create nonlinear classifiers by applying what is called a kernel trick to maximum-margin hyperplanes[11]. The current standard of SVMs uses soft margins and was proposed by Corinna Cortes and Vladimir Vapnik[12]. An SVM that uses only linear classifiers is one of the earliest learning models in Machine Learning[13].

Despite being one of the first machine learning algorithms it has qualities, such as robustness and simplicity, that make it preferable in some cases to newer classifiers like ANNs or, more recently, deep learning networks. It is one of the most efficient machine learning algorithms, with properly selected parameters it can offer a decent level of accuracy, it is simple to use and fairly fast, even with large datasets. It is naturally resistant to overfitting and is suitable for a variety of tasks, such as facial and speech recognition, image recognition, and text categorization[14].

2.5 Input/Output operations in parallel computing

Traditional computing uses serial I/O to store data from an application to the file system or another form of repository. This includes moving data to an output like a hard drive or moving data into memory from a hard drive. The two most common I/O operations are read and write and are similar in execution cost although there are some differences. In serial I/O read, several programs can still read from a file concurrently, while only a single program can write to a file at a time, thus avoiding concurrent writes overwriting each others data[15].

Parallel computing also uses serial I/O. When doing read operations, multiple processes can all open the same file and read from it as read operations do not alter the content of a file, the file is opened and an iterator moves from the beginning of the file to the end, reading data as it passes over it. The write operation similarly allows only one process to write to a file at a time. This is done to preserve the integrity of the data and to prevent multiple processes from overwriting each others data[16] [15].

With increasing size of data sets and the availability of more and more processors, serial I/O has become a bottleneck and the need for parallel I/O has become more obvious. New methods had to be developed to increase data access and especially the write speeds of serial I/O, one approach is to perform writes on multiple hard drives concurrently. Higher level solutions include parallel file systems like IBM GPFS[17], PVFS[18], and Lustre[19]. These parallel file systems distribute data across multiple servers and providers to enable concurrent access to files[15].

2.6 HDF5

HDF5 is a Hierarchical Data Format and a successor to HDF (HDF without any versioning refers to version 4.x and previous releases)[20]. It is designed to address some of the limitations of the older HDF product and to address current and anticipated requirements of modern HPC systems and applications. Its goal was to meet the ever-increasing demands of scientific computing and to take advantage of the ever-increasing capabilities of computing systems[21].

HDF5 consists of a data format specification and a supporting library. The data format has two primary structures, groups and datasets, and various secondary properties. Groups behave similar to directories in *NIX systems, datasets are N-dimensional arrays of data, and attributes are annotations that can be put on both groups or datasets. The library is implemented in C and provides routines for creating HDF5 files, creating and writing groups, datasets and attributes to HDF5 files, and reading groups, datasets, and attributes from HDF5 files.

Wrappers are then provided to allow usage of the library with other programming languages like FORTRAN 90, C++, Java, Python and MatLab[20].

2.7 PiSVM and its I/O

PiSVM is a parallel SVM implementation, implemented in C++, that aims to increase scalability beyond the original LibSVM it is based on.

The PiSVM[22] this thesis is based on is unversioned but it is a fork of PiSVM 1.2 that is maintained at JSC[23]. Since PiSVM is based on LibSVM they both use the same input format and command line switches. The three parts of PiSVM are Train (creates the model), Predict (classifies data) and Scale (brings data values into ranges of [-1, +1] or [0, 1] for easier and more accurate calculation). This thesis adds a fourth component, named PiSVM-Parse, to PiSVM. This tool will be used to convert data files between formats.

Currently, PiSVM only supports the text-based SVMLight file format, for storing both data and model.

Listing 2.1: SVMLight descriptor

```
1 <line> .=. <target> <feature>:<value> ... <feature>:<value> \# <info>\newline
2 <target> .=. +1 | -1 | 0 | <float>\newline
3 <feature> .=. <integer> | "qid"\newline
4 <value> .=. <float>\newline
5 <info> .=. <string>\newline
```

Each line in the file then looks similar to this

Listing 2.2: SVMLight sample data

```
1 5 1:0.43 3:0.12 9284:0.2 # Comment text
2 9 1:0.8 2:0.96 28:0.3 # Comment text
```

The first value in the line is the label (supervised data), then come featureID:value pairs and finally an optional comment. FeatureID:value pairs are 1-based, absent values are only filled in if the -D (dense format) flag is provided during training, they are assigned the value of 0.

A thorough analysis of the PiSVM code reveals that each MPI process, assigned to execution of PiSVM, reads the complete training data file through twice. Once to count the number of data elements and number of lines in the file, and then after allocating storage space, it reads the file again and parses featureID:value pairs into memory. After the model has been constructed only the first MPI process is responsible for writing the model to disk.

Further analysis reveals that during prediction each process first reads its own copy of the model file into memory. Then the whole data file is read through by all the processes to find the total number of lines. After that, the task of classifying lines is assigned to each process by dividing the number of lines between the processes. Then the data file is again read through from start to finish and each process classifies only the lines assigned to it.

2.8 Extending I/O in PiSVM

The programming part of adding the HDF5 support should require little alteration to existing code and not much MPI programming.

Initial evaluation of the PiSVM code suggests only a limited speed increase in training/testing since the training datasets are not very large and each process has to read the entire data into memory. The HDF5 code will read the data into a similar struct in memory and then have pre-existing code process it, and write the result to an HDF5 model file.

PiSVM-Predict is more complicated as the SVMLight read and processing code is interwoven. The data is read line by line and immediately processed with the help of the model file. In the HDF5 implementation, data will be read in chunks from the data file into arrays in memory. The existing processing code will have to be

replicated and altered so that it can process arrays in memory instead of individual lines read sequentially. The model file is read in the same way between versions. Tests will then have to be run to certify that the processing has not changed and accuracy stays the same in both implementations.

The file layout will have to be designed with I/O speeds in mind. HDF5 offers almost unlimited complexity and size because of its hierarchical support but for this project, the layout will have to be kept fairly simple. It is desirable to keep datasets and attributes as close to the root element as possible to simplify access.

HDF5 offers compression, and even moderate amount of compression can reduce the size of the file by half. The default compression will have to be chosen by testing read speeds of HDF5 files in both Train/Test and Predict phase. It is difficult to predict how the size of data files will change but hopefully, a modest size reduction will be observed. This is a secondary concern but a nice benefit.

HDF5 offers chunking in combination with compression. This will be beneficial when reading large data as each process will only have to read the chunks that contain the data it is responsible for, and not the whole data set.

SVMLight formatted files are larger and thus it takes longer to read them, they have to be read twice through, once to count elements and lines in order to allocate memory and prepare the data structure for receiving data read during the second pass.

One of the end products of this project is a parser that can translate data from the text-based SVMLight format into the binary based HDF5 format with a data structure designed around speed increase over every other requirement, a secondary objective was storage space reduction by using compression that HDF5 provides. An optimal balance of speed increase vs compression was desired. This parser does add execution time, based on the size of the data being parsed, but that is a one-time added execution time, as data is parsed into a format that then should not need alteration.

The other end product is a version of PiSVM that supports the HDF5 binary file format and can read data and write the model during training/testing, and read the data during prediction.

2.9 Summary

Remote sensing is the process of making observations of various objects such as houses, trams, cars, trees, and forests, etc. RS data is becoming more granular and detailed, this creates larger datasets that need to be processed and then classified with ML algorithms that run in parallel on HPC clusters.

HPC offers good performance with many cores and fast communication between those cores via a dedicated interconnection. An alternative is HTC where data is distributed between nodes that are more independent than HPC hardware. HTC tasks are focused on each node doing localized work, with less need for inter-node communication.

Supervised learning is the task of training a model from labeled training data. Unsupervised learning trains a model from unlabelled training data. The third learning model is called reinforcement learning, where software is taught to take actions as to maximize some notion of cumulative reward.

Support Vectore Machines is a supervised Machine Learning algorithm for classifying labeled data. Other ML algorithms include Artificial Neural Networks and deep learning networks. SVMs still use linear (or near linear) I/O operations and this prevents the use of larger datasets.

PiSVM is the HPC software suite chosen to showcase the implementation of an improved I/O framework design for parallel SVM implementations in general. PiSVM currently uses text-based files for data which are large and need to be read through twice.

Multiple data frameworks exist that are used for storing specific types of data or used in specific fields but those are not useful in this case. Two general purpose data frameworks exist that can be integrated into PiSVM, HDF5, and NetCDF, NetCDF has fewer features and does not support data hierarchy as complex as HDF5 does, so HDF5 was chosen for this master thesis. The file layout will have to be carefully designed and tested so that it conforms to the remote sensing application goals of improving I/O speed as much as possible.

3 Related work

Some Machine Learning software supports at least HDF, but most do not support HDF5 yet[24].

- MLPACK 3.0.0 (HDF)
- Indefinite Core Vector Machine 0.1 (HDF)
- SHOGUN 4.0.0 (HDF)
- OptWok 0.3.1 (HDF)
- Apache Spark (HDF5) beta version of a HDF5 connector, developed by the HDF Group[25]

The author is not aware of work beind one in adding advanced I/O framework support to SVM implementations other than PiSVM and the CascadeSVM implementation described below. However, the way the advanced I/O framework and the HDF5 support was implemented, it would be relatively straightforward to implement it in other libraries, such as LibSVM.

3.1 CascadeSVM

A student at the University of Iceland, Kveldúlfur Þrastarson, is working on the design, analysis, and implementation of a parallel and scalable cascade support vector machine framework[1](CascadeSVM[26]). The I/O framework developed in this thesis is re-used in the implementation of CascadeSVM so that it will have HDF5 support.

CascadeSVM works differently than PiSVM in that it takes the data and divides it up into roughly equal parts, and then each part is handed to a process that trains on the data and delivers a model file containing the SVs chosen from that part. The models are combined in pairs into a single problem that is then processed again (cascaded) until only a single model file is produced. The advantage of this method is increased speed, especially with large datasets, as kernel computation is faster with smaller pieces of data. The disadvantage of this approach is that valid SVs can

be discarded early in the cascades, thus leading to reduced accuracy. The reduction in accuracy is considered acceptable for the speed improvements offered.

3.2 Common Data Format - NetCDF

NetCDF is the biggest contender next to HDF5 in terms of flexibility, as other formats tend to be focused towards storing specific types of data (Raster, geospatial data, Neuroscience data etc). NetCDF is a set of data formats, programming interfaces, and software libraries that help read and write scientific data files. The data model consists of variables, dimensions, and attributes. Variables are N-dimensional arrays of data, dimensions describe the axes of the data arrays and attributes annotate variables or files with small notes or supplementary metadata[27].

NetCDF does not support structuring data in a hierarchical form like HDF5 does. This lack of features, like support for groups, is the main reason it was decided not so use NetCDF for this thesis. Since version 4.0 (2008) NetCDF can read and write HDF5 files, although with limitations, so applications that only support NetCDF will be (to some extent) able to use the HDF5 formatted files if their layout is designed with this limitation in mind.

3.3 ROOT

ROOT is an object-oriented library that was developed by CERN[28]. It was originally designed to contain data related to particle physics analysis, and the library reflects this by incorporating functionality beyond data storage, such as being able to do matrix algebra or curve fitting. ROOT is also used in other applications such as astronomy and data mining.

It has been criticised for its size and code bloat, weird inheritance class structure, massive method count (over 300), liberal use of global variables, and no separation of data and presentation[29].

Data presentation seemed to be a larger part of ROOT than data storage, because of that, and other criticisms it has received, it was decided not to use it for this thesis.

3.4 Apache Spark

Apache Spark[30] is an open-source cluster-computing framework that seeks to extend the capabilities of Apache Hadoop[31]. Hadoop provides a software framework for the processing of big data using a MapReduce model. Spark was developed to address limitations in Hadoop and MapReduce. It can be programmed for much more complex classification tasks. MLlib[32] is Apache Spark's scalable machine learning library. It only offers support for linear SVMs and does not support nonlinear classification[33].

Apache Spark supports a wide variety of distributed storage, including Hadoop Distributed File System[30], Cassandra[34], OpenStack Swift[30] and more, but the data files will need to be imported into those distributed storages before being usable in either Hadoop or Spark. SVMLight can be read into distributed storage and HDF5 is supported via HDF5 Connector for Apache Spark[25], which is currently in beta.

3.5 GPU-accelerated LibSVM

Some libraries have been written specifically to improve some parts of LibSVM. PiSVM, for example, uses MPI to run LibSVM in parallel on HPC clusters. Other libraries use GPUs to improve performance on the kernel computation part of LibSVM. These libraries are commonly referred to as GPU-accelerated LibSVMs.

Two of those are ThunderSVM[35] and GPU-accelerated LibSVM[36].

ThunderSVM is an efficient and open source SVM software toolkit which uses GPUs and Multi-core CPUs to accelerate kernel computation. It uses GPUs through the CUDA framework. It offers the same functionality as LibSVM. ThunderSVM only supports reading data in SVMLight format.

GPU-accelerated LibSVM is a modification of the original LibSVM and uses the CUDA framework to significantly reduce processing time. As it is a modification of LibSVM it offers the same functionality. GPU-accelerated LibSVM only supports reading data in the SVMLight format.

Both of those libraries use the CUDA framework[37], which is a proprietary computing platform and programming model that enables general computing on GPUs. CUDA works in conjunction with GPUs made by NVIDIA and this facilitates vendor lock-in. The benefits of GPU-acceleration is that it allows developers to run sequential parts of the workload on CPUs while the computation intensive portion of the

application is run on GPU cores in parallel. CUDA supports popular languages such as C, C++, Fortran, Python, and MATLAB.

3.6 Summary

Little work has been done in bringing high performance, parallel, and scalable I/O frameworks to HPC libraries. HTC systems usually offer their own version of distributed storage but those HTC systems support very few ML algorithms and often only support linear implementations of popular tools like SVM. CascadeSVM offers the promise of a considerable increase in execution speed at the cost of reduced accuracy by dividing the data up into pieces to make kernel computation faster. NetCDF only offers a subset of the functionality of HDF5 and because of that, it was discarded as a candidate for this thesis. GPU-accelerated libraries are becoming more common but rely on a proprietary framework and specific brands of GPUs in order to work, this leads to vendor lock-in.

4 Architectural design and Implementation

4.1 Overview of original functionality in PiSVM

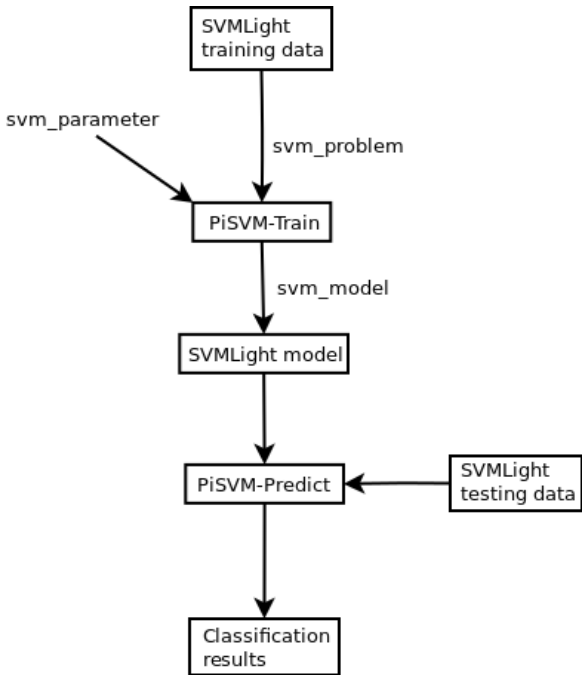

Figure 4.1: Process flow through PiSVM before parallel I/O framework was added.

Figure 4.1 illustrates the original process flow of the PiSVM tools. Training data is read into PiSVM-Train and into structs (svm_parameter and svm_problem) that are kept in memory. From the data in those structs a model is trained (svm_model struct) and written from memory to disk. The model and testing data are then both read by PiSVM-Predict, which proceeds to classify testing data based on information from the model. The testing data is labelled and PiSVM-Predict can track how accurate its predictions are. Those predictions and the accuracy are then written out to disk.

4.2 Overview of thesis additions

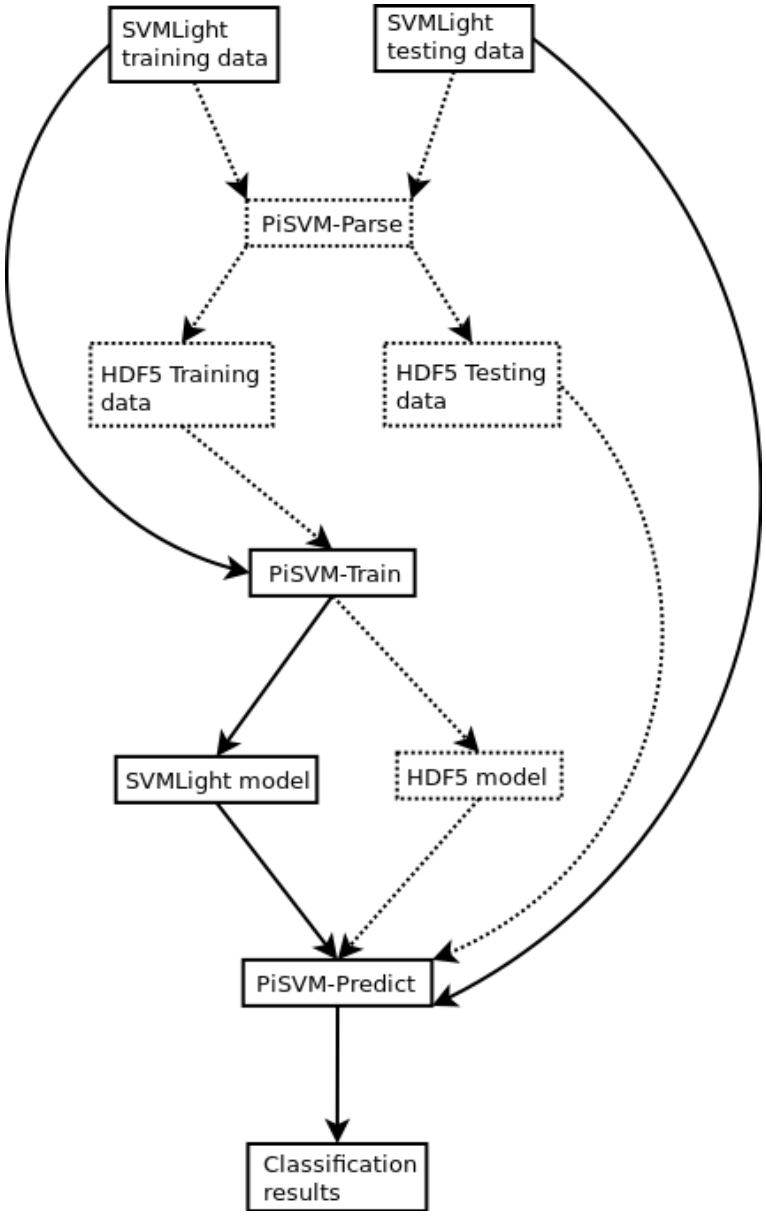

Figure 4.2: Process flow through PiSVM after parallel I/O framework was added.

Figure 4.2 shows the existing parts of PiSVM (solid) and the parts that this thesis added (dotted). To the PiSVM suite of tools, a parser was added and named PiSVM-Parse. It reads data in SVMLight format and writes out a file in HDF5 format. Functions to read and write HDF5 data were added to PiSVM-Train and PiSVM-Predict.

4.3 Data manipulation and storage

The two main parts of the PiSVM suite are PiSVM-Train and PiSVM-Predict. Train reads training data and creates a model by using SVMs. Predict tests the model on test data and provides an accuracy rating of the model, based on the amount of correctly classified data.

In most cases, data in PiSVM is read into C/C++ structs that are either passed between functions as parameters or stored as global parameters until the program finishes execution. The exception to this is in PiSVM-Predict where data is read and classified sequentially without being retained in memory.

The struct svm_model is used to contain data created by PiSVM-Train. It is read in by PiSVM-Predict from a .model file and used in predicting the classification of data from the test dataset. It contains variables concerning the training of the model, what parameters were chosen, number of classes, number of support vectors, and the feature:value pairs found in each support vector. When a model file is read by PiSVM-Train the variables label, nSV and free_sv are used.

Listing 4.1: svm_ model structure

```
1  struct svm_model
2  {
3    svm_parameter param;   //parameter struct
4    int nr_class; //# of classes, 2 in regression/one class svm
5    int l;           //total #SV
6    Xfloat **SV;     // SVs
7    int **nz_sv;     // SV featureID
8    int *sv_len;     // number of features in SV
9    int max_idx;     //highest featureID found
10   double **sv_coef;//coeffs for SVs in decision functions (
         sv_coef[n-1][l])
11   double *rho;    //constants in decision functions (rho[n*(n-1)
         /2])
12   double *probA;    // pariwise probability information
13   double *probB;
14
15 // for classification only
16   int *label;    // label of each class (label[n])
17   int *nSV;    // number of SVs for each class (nSV[n])
18   // nSV[0] + nSV[1] + ... + nSV[n-1] = l
19
20   // XXX
21   int free_sv;   //1 if svm_model is created by svm_load_model
22           //0 if svm_model is created by svm_train
23 };
```

The struct svm_problem is used to store the training data in memory and is used by PiSVM-Train to train the model in accurately predicting data. It contains variables that store information about the dataset, such as number of lines, the labels of each line, the feature ID's found and their corresponding values. Also necessary to store is the highest featureID found, as that is used to create dense data.

Listing 4.2: svm_problem structure

```
1 struct svm_problem
2 {
3   int l;            //number of lines in data file
4   double *y;        //label
5   Xfloat **x;       //typedef float Xfloat, feature values
6   int **nz_idx;     //featureIDs
7   int *x_len;       //# of features
8   int max_idx;      //highest featureID found
9 };
```

The struct svm_parameter is used to store parameter configurations chosen when a model is trained. Parameters like the type of SVM or type of kernel are written into the .model file at the end of training.

Listing 4.3: svm_parameter structure

```
1  struct svm_parameter
2  {
3    int svm_type;       //Type of SVM
4    int kernel_type;    //Type of kernel
5
6    int degree;         //for poly
7    double gamma;       //for poly/rbf/sigmoid
8    double coef0;       //for poly/sigmoid
9
10   /* these are for training only */
11   double cache_size;   //in MB
12   double eps;          //stopping criteria
13   double C;            //for C_SVC, EPSILON_SVR and NU_SVR
14   int nr_weight;       //for C_SVC
15   int *weight_label;   //for C_SVC
16   double* weight;      //for C_SVC
17   double nu;           //for NU_SVC, ONE_CLASS, and NU_SVR
18   double p;            //for EPSILON_SVR
19   int shrinking;       //use the shrinking heuristics
20   int probability;     //do probability estimates
21   int o;
22   int q;
23 };
```

The structs are central to the way the PiSVM tools operate. So as not to alter any parts that could affect accuracy, the functionality of SVMLight read and write functions will have to be replicated. HDF5 functions will be written that are able to read data from HDF5 files into corresponding structs, and write out data from structs into HDF5 files. This will ensure that accuracy stays the same. For those areas where changes need to be made (PiSVM-Predict) tests will the be run with HDF5 and SVMLight files as inputs to verify that accuracy has not changed.

4.4 HDF5 file structure

HDF5 is a hierarchical data format that can store complex and large datasets. A file structure for both the data file and model file will have to be designed. These file structures will need to reflect the overall thesis goals of improving I/O time, so the file elements should be kept as close to the root element as possible to reduce access times. HDF5 offers various elements like the file object, groups, custom and standard attributes, datasets, compound datasets, links, and datatypes.

The file object is the root element and can contain datasets and groups. Groups are similar to directories in *NIX type systems and divide the file up into parts. Both the file object, groups, and datasets contain standard attributes that describe properties like size. Standard attributes are not alterable through the API but are updated to reflect changes to the object they are attached to. Custom attributes can be created, modified and deleted by the API and can store additional information.

Normal datasets store atomic and native datatypes, compound datasets store data represented with compound datatypes. Compound data types are made up of atomic datatypes and their design can be stored in the HDF5 file for portability.

4.4.1 HDF5 data file

Listing 4.4 illustrates the data file schema. It is used for both training and testing and is thus read by both PiSVM-Train and PiSVM-Predict. The data file is written by PiSVM-Parse. It was designed to be as flat as possible in order to minimize access time. The schema uses only native datatypes to ensure maximum portability. Using native datatypes does put the onus on the implementer to make sure that the data is read into datatypes that have at least as much accuracy as the native types.

Listing 4.4: HDF5 data file schema

```
File file
{
  //Attributes written in H5S_SCALAR attribute space
  H5::PredType::NATIVE_INT DenseFeatureCount; //32-bit integer
  H5::PredType::NATIVE_INT SparseFeatureCount; //32-bit integer
  H5::PredType::NATIVE_INT ChunkCount; //32-bit int, default 64
  H5::PredType::NATIVE_INT CompressionLevel; //32-bit int, def. 9

  //Datasets
  H5::PredType::NATIVE_DOUBLE class; //64-bit floating-point
  H5::PredType::NATIVE_UCHAR densitymap; //8-bit unsigned char
  H5::PredType::NATIVE_FLOAT feature_value; //32-bit float
};
```

Figure 4.3 illustrates a feature of HDF5 that is called Chunking. Chunking divides a file up into parts that can then be read individually into memory by a processor, instead of that processor having to read the entire file into memory as is the case with contiguous datasets. By calculating chunk size from line count it is possible to calculate what chunks each processor should read based on its rank.

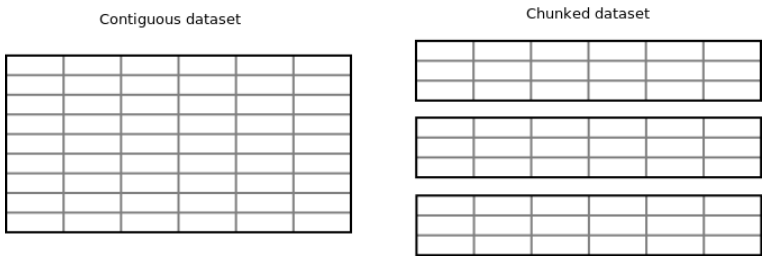

Figure 4.3: The difference between a contiguous dataset and a chunked dataset.

PiSVM-Parse takes as an optional parameter the desired ChunkCount, if no chunk count is provided then the default value of 64 is used. ChunkCount should optimally be a multiple of the processor count. If ChunkCount is 64 and 32 processors are used then each processor will read 2 chunks, if 16 processors are used then each one reads 4 chunks and so forth. When data is read from an HDF5 file only those chunks that are required are read into memory. This is less relevant in PiSVM-Train as the whole data file needs to be read into memory in order to train a model, and more relevant in PiSVM-Predict where each line can be classified in a linear fashion, thus reducing the amount of data that needs to be read will result in an increase in read speed.

PiSVM-Predict will need to know ChunkCount in order to calculate what lines each

processor should process and because of that ChunkCount is stored as an attribute.

CompressionLevel ranges from 0 (no compression) to 9 (maximum compression), a default value of 9 was chosen because that offers the best performance relative to reduction in data size. This was determined using timing data with varying levels of compression and their impact on speed, as explained in the Evaluation chapter.

During parsing, two variables named max_idx and line_count, are incremented as new features and lines are found. The variable line_count stores the number of lines and max_idx stores the highest featureID found while parsing the SVMLight file. These two values are used to create the datasets in the HDF5 file. Row count is equal to line_count and column count is equal to max_idx.

FeatureIDs need to be 0-based in svm_problem, while they are stored as 1-based in the SVMLight. When reading data from the HDF5 file, the column number is 0-based and thus corresponds to the way featureID needs to be numbered in svm_problem. The HDF5 API allows the dimensions of the datasets to be read so from the HDF5 file we can extract the values for line_count and max_idx in PiSVM-Train and PiSVM-Predict.

Dense data is constructed by reading the feature_value dataset into the struct svm_problem. The feature_value dataset contains all featureIDs from 0 to max_idx in their respective columns, with their corresponding values set to 0 whenever a featureID was not found in the SVMLight file. So when constructing svm_problem by reading dense data the whole feature_value array is just read straight into it.

Sparse data, however, is constructed by reading in only those values in the feature_value dataset whose corresponding value (line/column) in the densitymap dataset is 1 (true). An svm_problem struct created from reading sparse data will thus not have features that were not found in the original SVMLight file.

SparseFeatureCount stores the number of featureID/value pairs that were found in the SVMLight file during parsing.

DenseFeatureCount stores the count of all featureID/value pairs (even those not found in the original SVMLight file) and is a multiple of max_idx and line_count. DenseFeatureCount is used when the -D flag (Dense data) is provided when running PiSVM-Train.

The datasets all have the same row count, but column count varies. The class dataset only has one column and stores the class/label of the corresponding line. The datatype of 64-bit floating-point is used in the design because the label is stored as a double in svm_problem, the language descriptor, however, suggests using only a float datatype.

The densitymap dataset has 'max_idx' columns and contains 0 or 1 (false/true) depending on if that featureID exists in the SVMLight file and has a value. The data type of unsigned character was chosen as it is the smallest one available in HDF5 and it only needs to store the two possible values of 0 and 1, and those are easily typecast into boolean values.

The "feature_value" has 'max_idx' columns and contains the value of that particular featureID in that particular line. The native data type that represents 32-bit floating points is used because the value is stored as Xfloat in svm_problem.

4.4.2 HDF5 model file

The model file schema is designed to be as flat as possible, additional attributes are stored as metadata to track information that was used in the model file construction, such as SVM type and kernel type. The model file stores the output model from PiSVM-Train and used as input to PiSVM-Predict

Listing 4.5: HDF5 model file schema

```
File file
{
  //Attributes written in H5S_SCALAR attribute space
  H5::PredType::NATIVE_INT  FeatureCount; //32-bit int
  H5::PredType::C_S1        svm_type; //string
  H5::PredType::C_S1        kernel_type; //string
  H5::PredType::NATIVE_INT  nr_class; //32-bit int
  H5::PredType::NATIVE_INT  total_sv; //32-bit int

  //Datasets
  H5::PredType::NATIVE_DOUBLE SV; //64-bit floating-point
  H5::PredType::NATIVE_UCHAR  densitymap; //8-bit unsigned char
  H5::PredType::NATIVE_INT  label; //32-bit int
  H5::PredType::NATIVE_INT  nr_sv; //32-bit int
  H5::PredType::NATIVE_DOUBLE rho; //64-bit floating-point
  H5::PredType::NATIVE_DOUBLE sv_coef; //64-bit floating-point
};
```

The model file stores the Support Vectors (SVs) that are found while training. Each one of them has a label and a varying number of featureID:value pairs.
FeatureCount contains the number of featureID:value pairs.
svm_type stores the SVM type in string format (ex: c_svc).
kernel_type stores the kernel type used during training in string format (ex: linear).
nr_class stores the number of classes from the data file.
total_sv stores the total number of SVs found during training.

The dataset SV contains values for each SV, the column number represents the featureID and is 0-based.

The dataset densitymap has |max_idx| columns and contains 0 or 1 (false/true) depending on if a featureID belongs to a SV. SVs do not have dense data but since the SV dataset is essentially an array there might be featureIDs in the dataset whose value does not belong to an SV. The data type of unsigned character was chosen as it was the smallest one available, only two possible values are 0 and 1 and those are easily typecast into boolean values.

The dataset label only has one column and stores each SVs class/label for the corresponding line. The dataset nr_sv contains the number of SVs that belong to each class. The dataset rho contains bias terms in the decision function found during training. The dataset sv_coef stores the coefficients for each SV.

4.5 PiSVM-Parse

An integral part of the architectural design and toolset is a program for parsing data files from SVMLight into HDF5 format. The parser takes as parameters the input file name, and optional parameters chunk count, compression level, and the output file name. If omitted the default chunk count is 64, default compression level is 9 and the default output file name will be the input filename with the .el ending removed and .h5 appended instead.

An example command line that parses data from an SVMLight file into HDF5 format with data divided into 128 chunks, with a compression level of 9. As an output file name is omitted the output file will have the same name but with a .h5 ending instead of the .el ending.

Listing 4.6: PiSVM-Parse sample parameters

```
1 srun ./pisvm-parse -c 128 -z 9 sdap_area_all_training.el
```

The parser reads in each line from the SVMLight file, extracts the label at the start of the line, then extracts the featureID:value pairs until the end of the line is reached. It also tracks various parameters like total featureID/Value pair count for sparse and dense data, the chunk count and compression level, and saves them in the HDF5 file as attributes that can be read and examined later.

Once the parser has finished processing the SVMLight file it outputs a message along with the ChunkCount and Compression parameters that were saved to the HDF5 file.

4.6 Read and Write Functions

PiSVM-Train reads the parameters from the command line and stores them in a global struct svm_parameter. It then reads the data in from an SVMLight file into a global struct svm_problem that is stored in memory. PiSVM-Train also writes the content of the struct svm_model to a model file on disk.

A function was added to PiSVM-Train that reads the HDF5 data file and constructs an svm_problem struct that is identical to the one created by the SVMLight read function. The svm_parameter struct is read by another function that did not need much alteration except to deal with the -f parameter that describes the input file format.

An example command line with parameters to train a model on the Indian Pines Raw dataset. The parameter -f 1 instructs PiSVM-Train to treat the input file as an HDF5 formatted file.

Listing 4.7: PiSVM-Train sample parameters

```
1 srun ./pisvm-train -D -o 1024 -q 512 -c 100 -g 8 -t 2 -m 1024
2       -s 0 -f 1 sdap_area_all_training.h5
```

A function was also added that writes the contents of svm_model into the designed HDF5 format and writes it to disk.

PiSVM-Predict reads the model file into the svm_model struct, a function was added that reads HDF5 data and constructs svm_model that is identical to the one created by the SVMLight read function.

The main processing logic in PiSVM-Predict, that both reads the data and processes it, was replicated and altered so that it reads the data in HDF5 format into arrays allocated on the heap. The data in the arrays is then processed in a similar way as the SVMLight data is processed. Once processing is done the results are written to a file.

4.7 Architecture and integration challenges

Few challenges were encountered when implementing HDF5 support. HDF5 documentation is sparse and often auto-generated, with few code samples, often written in other languages, like Python.

Each processor in PiSVM-Train has to read the entire data set into memory because SVs are chosen relative to all the data points, so in order to find the best SVs, each processor must be aware of all data. If PiSVM-Train split the data up into chunks and had each processor then chose the best SVs from only the chunk it was assigned then the hyperplanes and the SVs would not be globally optimal, but rather locally optimal, and that gives subpar results. This would result in faster training but lower accuracy. This results in little room for improvement except in the speed increase from just switching formats. It is not possible to parallelize the read.

When writing the model file in HDF5 format there was an unexplained speed decrease when using compression, up to a thousandfold (write speed recorded at 53 seconds vs 48 milliseconds), as a result compression was disabled and the file size was thus only partially reduced.

No problems were encountered in PiSVM-Parse as it responded well to optimization.

4.8 Summary

When using SVMLight the data is read from the file and into structs that are stored in memory while the program executes. Functions were written for HDF5 that replicate SVMLight read and write functionality. Data is read from the HDF5 file and into the same structs that are then processed by making inherent use of the LibSVM library. File schemas had to be designed for both the data file and model file. The data file schema is designed to be flat and as close to the root element as possible, in order to minimize access times. The model file schema was designed to store the same data as was stored in the SVMLight model file. Additional data, like number of features, was stored as attributes in the HDF5 to further reduce file read time. Chunking and compression are enabled for the data files, allowing PiSVM-Predict to read only the chunks containing the data it is responsible for classifying. More work was put into adding HDF5 support to PiSVM-Predict as a substantial amount of rewrite had to be done in order to read and process HDF5 data. The use of chunking and optimization of read operations is very important since in most cases testing data is bigger with regards to volume. A common training/testing split is 1/10th for training and 9/10th for testing. A parser was added to the PiSVM suite to convert data from SVMLight into HDF5 format.

5 Evaluation

5.1 Evaluation environment

Evaluation was performed on the HPC cluster JURECA at JSC[38] [39]. The specification of this cluster is as follows:

- Hardware
  - Base Machine
    - 1884 nodes
    - 2x Intel Xeon E5-2680 v3 Haswell processors per node
    - 12 cores per processor
    - 24 cores per node @ 2.5Ghz
    - 45,216 cores
  - Booster (KNL) Module
    - 1640 nodes
    - 68 cores per node
    - 111,520 cores
  - Total: 156,736 cores (151,040 measured cores)
  - Total: 287,744 GB of memory
  - Total: 1,345.28 kW (Submitted)

- Software
  - CentOS 7.x operating system
  - Batch system based on SLURM/Parastation
    - SLURM (workload management and UI)
    - Parastation (Resource management)
  - Software packages loadable as modules, necessary for evaluation were
    - Intel compiler
    - IntelMPI
    - HDF5

Sample submit script

Listing 5.1: Slurm Batch job

```
#!/bin/bash -x

#SBATCH --job-name=pisvm-train
#SBATCH --ntasks=64                 #run on how many cpus
#SBATCH --mem=600mb                 #memory limit
#SBATCH --output=%j.mpi-out         #where stdout stream goes
#SBATCH --error=%j.mpi-err          #where stderr stream goes
#SBATCH --time=00:30:00     #stop if job goes for longer than this
#SBATCH --partition=batch    #partition to use

# -f 0 is svmlight format, -f 1 is hdf5
srun ./pisvm-cyclicDistribution/pisvm-train -D -o 1024 -q 512
       -c 100 -g 8 -t 2 -m 1024 -s 0 -f 0 indian_raw_training.el
```

The SBATCH parameters set variables that Slurm uses when running the job. Those parameters also control where in the queue a job lands, if not enough resources are available to run the job then it gets pushed back until resources become available.
ntasks sets the number of processes to allocate for the job.
mem sets the amount of memory that the job requires.
Output and error parameters direct the stdout and stderr streams to the files specified here, %j indicates the job ID.
Time sets the maximum runtime, a job is cancelled if it goes over the specified time.
Partitions are part of the cluster setup and are used for different types of tasks, some partitions allow more memory or time, others allow access to GPUs.

5.2 Datasets for evaluation

The dataset chosen for evaluation was the biggest readily available, both in terms of bytes, and complexity, the Indian Pines raw dataset[40]. It does contain fewer lines of data than the Rome[41] dataset but it contains more classes and more features, making it more computationally complex. This results in a relatively high processing time over I/O time but there should still be considerable improvements to I/O operations.

Accuracy has been verified to stay consistent as support for HDF5 is added. There is still going to be a speed difference between the default and optimal parameters in PiSVM as they have an effect on what SVs are included in the model. Because of this, the optimal parameters that have been analyzed by Cavallaro and Riedel et al.[42], will be used throughout the evaluation and will not be changed between runs

to preserve consistency.

The Indian Pines dataset was gathered by AVIRIS sensor over the Indian Pines test size in Indiana. The Indian Pines scene contains two-thirds agriculture, and one-third forest or other natural perennial vegetation. There are two major dual lane highways, a rail line, some housing and other structures, and smaller roads. The dataset consists of 1417x614 pixels and 200 spectral reflectance bands (stored as attributes) in the wavelength range of $0.4-2.5x10^{-6}$ meters. The original dataset contained 58 classes but this dataset (despite being labeled raw) contains 52 classes. Six classes have been discarded because of too few numbers of samples. It contains 33.396 vectors[40].

5.3 Evaluation setup

Time measurements were performed using C++ std::chrono high_resolution_clock. Total execution time of PiSVM-Parse was measured when converting the Indian Pines Raw Testing file from SVMLight into HDF5 format with compression levels ranging from 0 to 9 in steps of 0, 1, 3, 5, 7 and 9. Compression level of 0 is included as it is a valid HDF5 compression level but the datasets in the file are left uncompressed. Compression level of 1 is the lowest compression, compression level of 9 is the highest. Initial expectations are that working with uncompressed data will be faster since no time has to be spent on uncompressing the data before using it. But it was unknown how much this speed difference would be, as it was possible that the smaller file from compression would result in reduced I/O time as well.

For PiSVM-Train the effect of compression level on input read time was tested by running PiSVM-Train on the files produced by the PiSVM-Parse tests, the testing dataset is considerably larger than the training dataset but this is still a valid test to show how much training times change with different levels of compression.

In PiSVM-Train the output write time was measured to establish how long it would take to write a model file in HDF5 format.

In PiSVM-Train the effect of increasing process count was measured and broken up where possible into input read time, output write time, and total execution time. These measurements were conducted for both SVMLight and HDF5 files.

For PiSVM-Parse the effect of increasing process count was measured and broken up into input read time, output write time, processing time, and overall execution time. These measurements were conducted for both SVMLight and HDF5 files. During these evaluations, the change in file size was measured in bytes.

The deciding factors in determining the default compression level for the parser are the time it will take to parse a file at various compression levels and if the compression levels have a huge effect on the read speeds of PiSVM-Train and PiSVM-Predict. A secondary factor is a reduction in file size.

Figure 5.3 shows that there is not a huge difference in input read time between compression levels, and figure Figures 5.2 shows that there is not a huge size difference in the data files between compression levels 1 and 9 either, this is because the dataset is relatively sparse on datasets that benefit from compression, only the 2d dataset feature_value responds well to compression. Because of this it was decided to set the default level of compression to 9. As dataset sizes continue to grow this decision might need to be re-evaluated.

5.4 Evaluation parameters

The following parameters were used during testing of PiSVM-Parse, PiSVM-Train, and PiSVM-Predict.

Listing 5.2: PiSVM-Parse evaluation parameters

```
1 pisvm-parse -c 64 -z {0-9} input_filename [output_filename]
```

-c is the ChunkCount parameter, using it allows programs to read only portions of the HDF5 file in chunks into memory. PiSVM-Predict uses this, as each process only reads the data it is responsible for classifying. The default value is 64.
-z parameter controls the compression level. 0 is uncompressed, 1 is the lowest compression level, and 9 is the highest compression level. The default value is 9.
input_filename indicates the file to be read, in SVMLight format.
output_filename is an optional parameter that indicates the output file name. If it is omitted then the input filename is used but the .el ending (if it has that) is replaced with a .h5 ending.

Listing 5.3: PiSVM-Train evaluation parameters

```
pisvm-train -D -o 1024 -q 512 -c 100 -g 8 -t 2 -m 1024 -s 0
  -f {0-1} input_filename [model_filename]
```

-D means all feature vectors should be read as dense, whenever there are features missing they are added with a value of 0.
-o sets the max size of working set.
-q sets the max number of new variables entering the working set.
-c sets the cost value of C for the SVM type C-SVC
-g sets the gamma value used in the kernel function.
-t sets the kernel type, 2 is the RBF kernel.
-m sets the cache memory size in MB.
-s indicates the SVM type to use, 0 is C-SVC.
-f controls the format of the data and model file, 0 for SVMLight, 1 for HDF5.
input_filename is the name of the data file that contains the training data.
model_filename is an optional parameter that indicates the name of the output model file. If it is omitted then the name of the training file is used but the .model ending is added to it.

Listing 5.4: PiSVM-Predict evaluation parameters

```
pisvm-predict -b 0 -f {0-1} input_filename model_filename
  [output_filename]
```

-b controls whether to predict probability estimates, 0 means not to predict probability estimates.
-f indicates the format of the data and model file, 0 for SVMLight, 1 for HDF5.
input_filename is the name of the data file that contains the testing data.
model_filename is the name of the model file created.
output_filename is an optional parameter that indicates in what file the output results should be written, if omitted the output is written to the stdout stream.

5.5 Speed Evaluation - PiSVM-Parse

PiSVM-Parse was run on the largest SVMLight file available, the Indian Pines Raw Testing file (750 MB, exact size can be found in table 5.1), and the output file was written with varying levels of compression, ranging from 0 (uncompressed) to 9 (highest compression) in steppings of 0, 1, 3, 5, 7 and 9.

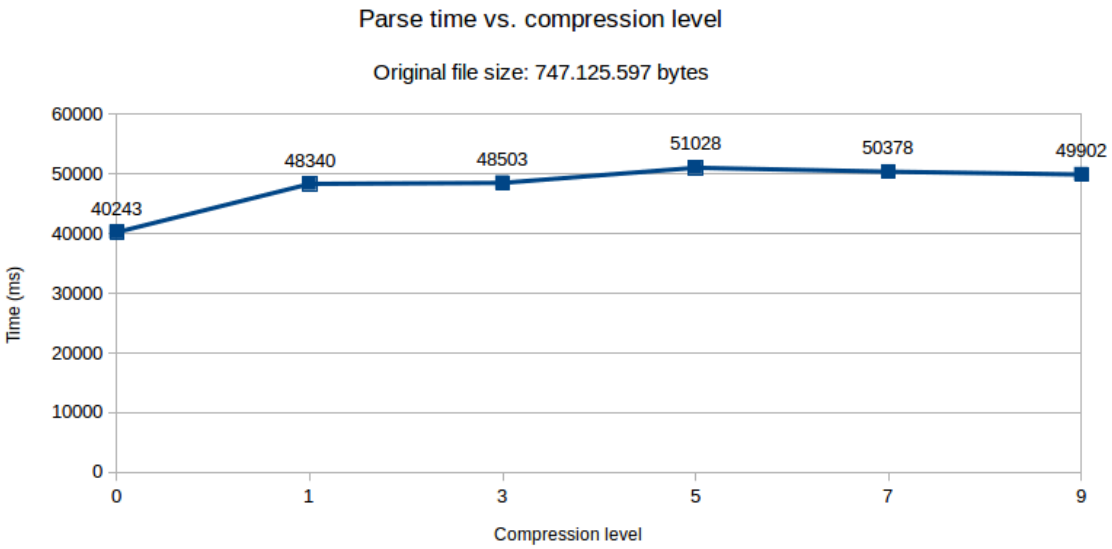

Figure 5.1: Parse - Changes to parse time as compression level is increased.

Figure 5.1 shows the total execution time of PiSVM-Parse as compression is increased. There is a 26.8% difference between the fastest and the slowest parse times, where the fastest time is without compression. The difference between the fastest and slowest parsing time with any level of compression is 5.6%. This shows that parsing is faster when the output is an uncompressed file. This is because no processing time has to be spent on compressing the datasets before they are written to the HDF5 file. Figure 5.3 shows that there is almost no speed penalty in input read speed as compression is increased. That, along with a reduction in size and that parsing should be relatively uncommon, suggests that that it is not beneficial to use less (or no compression) just to gain the speed increase during parsing.

File size was recorded for the original Indian Pines Raw Testing file in SVMLight format (Exact size can be found in table 5.1), as it was parsed into HDF5 format with varying levels of compression, ranging from 0 (uncompressed) to 9 (highest compression) in steppings of 0, 1, 3, 5, 7 and 9.

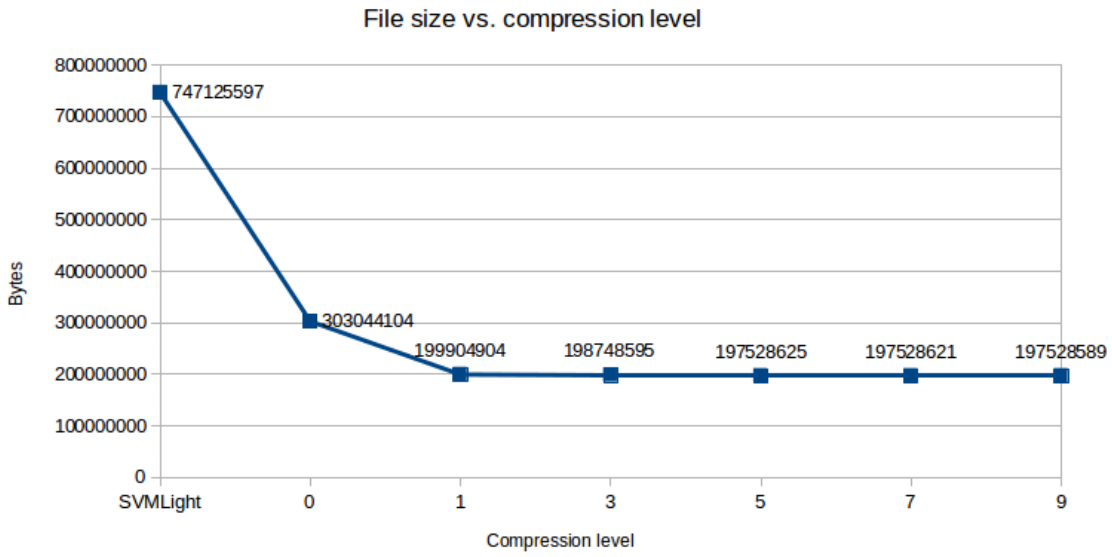

Figure 5.2: Parse - Changes to file size as compression level is increased.

Figure 5.2 shows changes in file size, from SVMLight format to HDF5 format with varying compression levels. A reduction in size by 59.4% is achieved just by changing formats. The greatest reduction in size is achieved with compression level 9, a reduction of 73.6%.

The greatest size reduction is achieved at the highest level but there is not a huge difference between compression levels 1 and 9, a total of 1.2% reduction. In this case this is explained by the fact that not all the datasets in HDF5 offer the potential for a lot of compression. Only the feature_value dataset has any real potential to be compressed as it is two-dimensional and stores 32-bit floating-point values. Densitymap is already small because of the choice of 8-bit unsigned character value and the class dataset is a single dimension dataset. They still have compression enabled, but they are just not that big compared to the feature_value dataset. As a consequence, the impact of the possibility of choosing these levels of compression become more relevant in simpler datasets, than complex remote sensing datasets, such as those used in retail or other commercial setups.

5.6 Speed Evaluation - PiSVM-Train with regards to compression level

In PiSVM-Train each process reads the entire file into memory and it is then processed. Once processing is done every process sends his results to the rank 0 process who then writes out the model file, this communication is handled by the psmo solver and is not a part of the base PiSVM code. So only the rank 0 process has the data in memory to write out the model file. This makes it harder to implement any sort of parallel writing as that would require substantial changes to the psmo solver.

Input times were all measured with the Indian Pines Raw Training data file (83 MB, exact size can be found in table 5.1). 16 processes were used to reduce processing time, as the number of processes does not affect I/O time. Measurements were done on both SVMLight formatted data files and HDF5 formatted data files, with varying compression levels, ranging from 0 (no compression) to 9 (maximum compression) in steppings of 0, 1, 3, 5, 7 and 9. The objective was to measure the input read time changes.

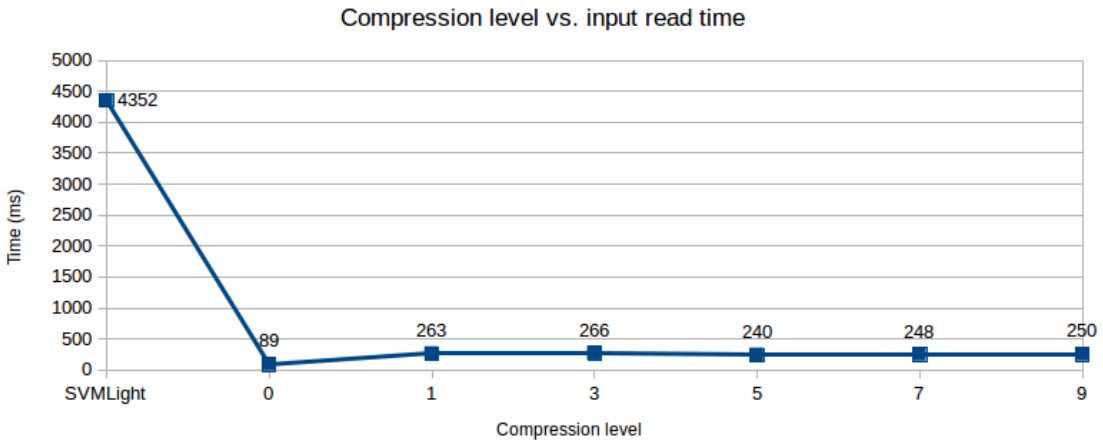

Figure 5.3: Train - Changes to input read times as compression level increases.

Figure 5.3 shows that a 97.95% speed increase is achieved by changing format from SVMLight to HDF5 with no compression. As compression is increased input read times do not increase by a large amount compared to other compression levels. There is a 94.2% reduction in input read time between SVMLight and HDF5 with compression level of 9.

With regards to compression level, there is a 9.7% difference between the fastest and

slowest input read time with any compression level (excluding compression level 0). While this is a decent amount in percentages, the variation of actual values is so small that this difference can be explained by varying amounts of load on the file system as measurements were being conducted. As the size of datasets increases this difference should be more noticeable.

The output write times were measured with the model file resulting from training with the Indian Pines Raw Training data file in both SVMLight format and HDF5 format. The resulting model file size is shown in table 5.1.

Only the first process writes out the model file and the file format is dictated by the input file format, an SVMLight data file results in an SVMLight formatted model file.

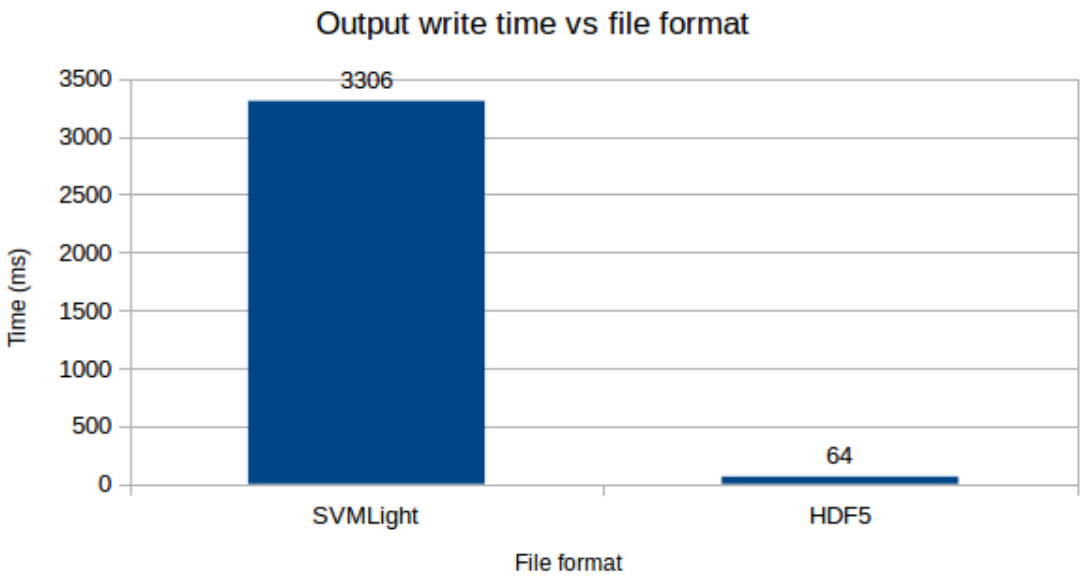

Figure 5.4: Train - Difference in output write times between file formats.

As illustrated in figure 5.4 the output write time is reduced from 3306 ms to 64 ms, a 98% reduction. This is achieved by using all the features that HDF5 offers and by utilizing a model file schema designed for speed and flexibility. The model file size has been reduced by about 24.4% (see table 5.1) so some of the speed increase is also because it is faster to read a smaller file.

Total execution time was recorded with the original Indian Pines Raw Testing file in SVMLight format (Exact size can be found in table 5.1), and in HDF5 format with varying levels of compression, ranging from 0 (uncompressed) to 9 (highest compression) in steppings of 0, 1, 3, 5, 7 and 9.

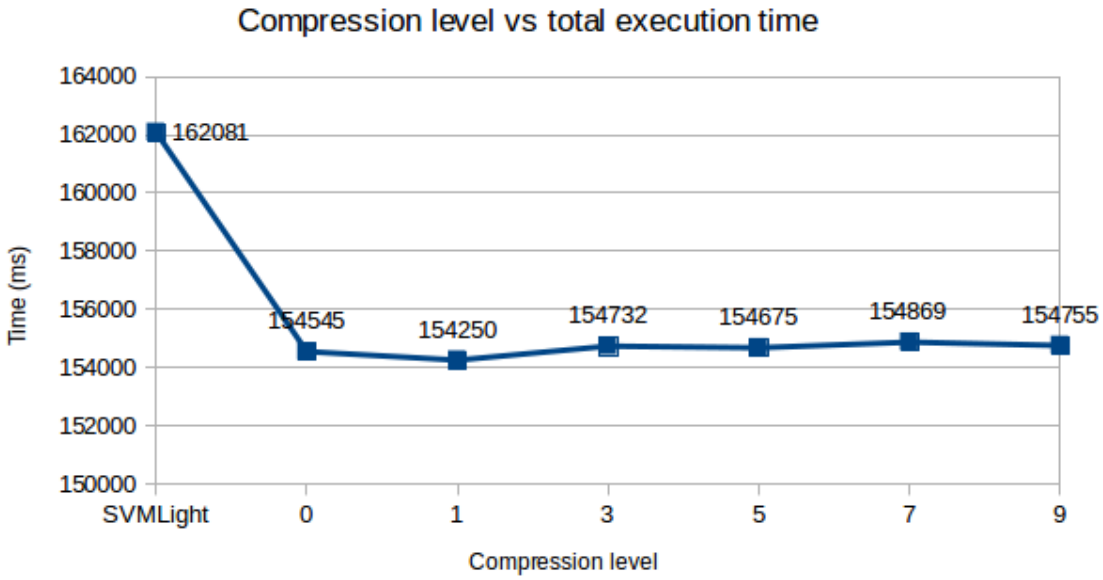

Figure 5.5: Train - Changes to total execution time as compression level is increased.

Total execution time is reduced by 4.5% by using an HDF5 formatted data file with compression level 9 over SVMLight data. There is not a big difference between total execution times with HDF5 files with compression levels between 1 and 9. Variations in execution time, when using HDF5 files as compression level increases, can be explained by variations in loads on the file system between executions.

5.7 Speed Evaluation - PiSVM-Train with regards to an increased number of processes

Changes in execution time in PiSVM-Train were measured as process count was increased. Input read times were all measured with the Indian Pines Raw Training data file in SVMLight format and HDF5 file format with compression level of 9. The files exact sizes can be found in table 5.1.

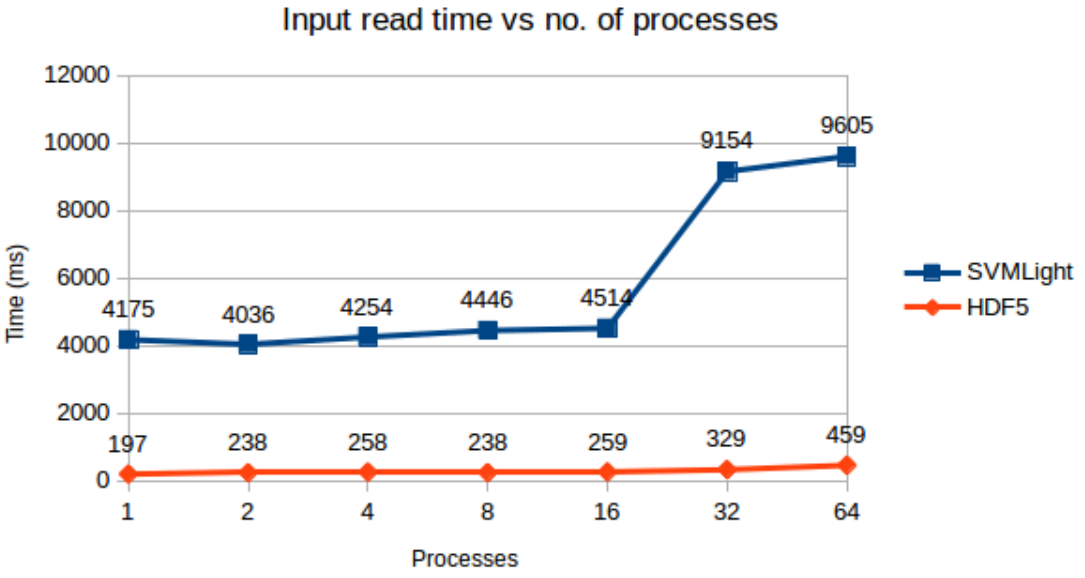

Figure 5.6: Train - Changes to input read time as number of processes is increased.

Figure 5.6 illustrates how input read time changes between formats as process count increases. An average reduction of 94.3% is achieved up to 16 processes. That is because all the processes read the entire data file into memory before processing it so increasing the number of process does not affect the speed by any noticeable amount. From 16 to 32 processes there is an increase in input read time for SVM-Light. Further testing reveals that this happens as process count moves beyond 24 processes. This is explained by the HPC architecture, each node has 24 cores and the leap most likely due to inter-node communication overhead. An increase is also observed with HDF5 but it is much less apparent because of the lower read times, this is most likely because of the HDF5 files smaller size and optimization done in the HDF5 framework.

Total execution time was recorded with the Indian Pines Raw Testing file in both SVMLight and HDF5 format as process count was increased.

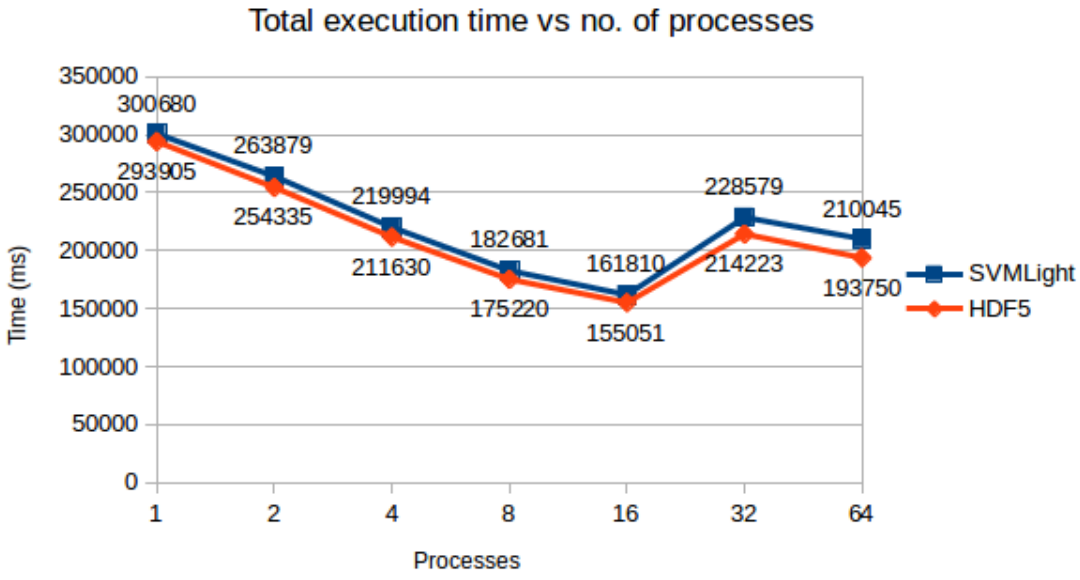

Figure 5.7: Train - Evaluation of execution time as number of processes is increased.

The total execution time is reduced, on average, by 3.45% as process count is increased. Figure 5.7 shows how the processing time dominates the total execution time, as the total execution time is not decreased tremendously using the two different file formats, despite the gains in input read speeds. The time it takes to process the data is very similar between the formats as the datasets contain the same vectors and all parameters are the same.

The increase in time between 16 and 32 processes happens at the 24 process mark. As with the changes to input read speed, the explanation is the architecture where each node contains 24 cores and this jump is because of inter-node communication overhead.

5.8 Speed Evaluation - PiSVM-Predict

Classification using PiSVM is highly scalable and responds well to parallelization because of the way data is classified. There is no inter-node communication done, it is enough to just read the model file and then that is used to classify data.

When doing classification on data that is read in SVMLight format, each process reads in its own copy of the model file and then they all open the data file and read it through once to get the line count. Each process then calculates what lines it should classify based on the total line count and its own rank, so that each process gets roughly the same number of lines. Once local upper and lower line numbers have been established each process reads the data file through again and classification can be performed.

When a process reads in a line it is not responsible for classifying, it gets read and parsed but then discarded. However when a process reads in a line it is responsible for classifying it gets classified and the results are stored. The actual class (test data is supervised) is then compared to the expected class and accuracy statistics are updated based on those results.

All lines get read by all processes, even if that process is not responsible for classifying that particular line. Once all lines have been processed, the results are communicated via MPI to the first process, which is then responsible for writing the output, accuracy and classification of each line. Unfortunately with the way SVM-Light data is classified, since data is read and immediately classified, this makes it harder to measure independently the input read time and processing time, as both operations are performed in quick succession. The way the code is structured makes it impossible to measure each time separately.

When using HDF5 however, one of the key elements of the architectural design of the I/O framework is that the line count can be retrieved immediately from the dataset. Each process then calculates what data it is responsible for classifying and, through chunking, reads only those lines into memory. The processing code then only classifies the data it has read into memory. This makes it possible to separately measure input read time and processing time.

The classification code was tested with an SVMLight data file, and the combined input read time and processing time were found to be consistent between runs when using the same data. Since the actual classification code is unaltered between the SVMLight and HDF5 implementations it is assumed that the processing times will be equal, then the input read time can be calculated for SVMLight and compared to the HDF5 input read time. This is a necessary assumption as it is not possible to separate the input read time and processing time when classifying data in SVMLight format. An alternative test was conducted where the HDF5 input read times and processing times were combined and then compared to the measured SVMLight time. Figure 5.8 illustrates the results of those tests.

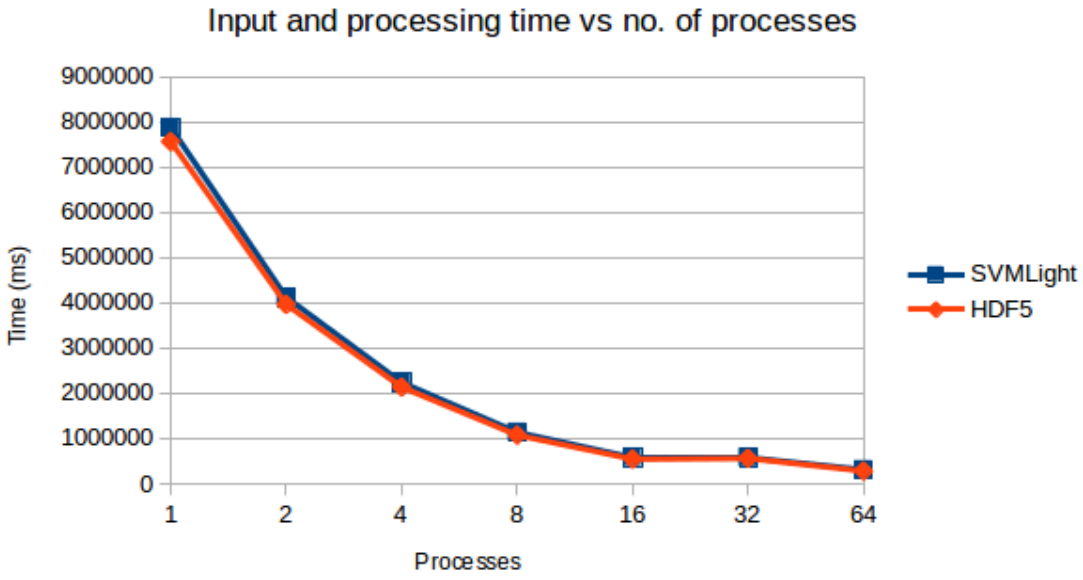

Figure 5.8: Predict - Combined input and processing time as number of processes is increased.

Figure 5.8 shows the measured time of SVMLight and the combined input read time and processing time of HDF5. An average reduction 3.95% in input and processing time was observed up to 16 processes. This does not seem like a huge improvement so it becomes necessary to examine only the input read times.

Figure 5.8 also illustrates how much read and processing times are improved by adding processes. This underlines how highly parallelizable the task of classification is.

The input read time of SVMLight was calculated by assuming equal processing time between SVMLight and HDF5. This makes the speed increase with HDF5 much more noticeable.

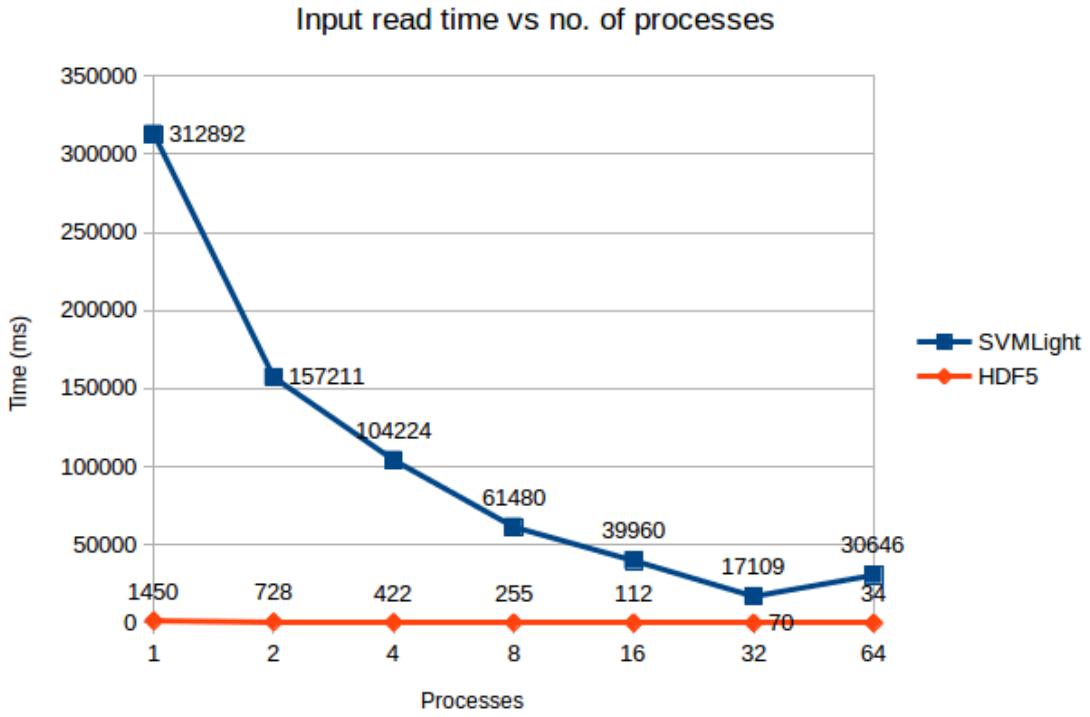

Figure 5.9: Graph shows the difference in input read speed between the two formats as process count increases. An average of 99.54% decrease in input read time.

The speed increase is both because of faster input read times as a result of the binary-based file format, but also because of chunking. Each process only has to read those chunks that contain the lines it is responsible for processing and everything else can be skipped.

Because of how dominant the SVMLight input read time is it is not apparent but in figure 5.9 there is also a very steep reduction in input read times for HDF5. The input read time almost halves as the number of processes is doubled. For example from 4 to 8 processes there is a 41% decrease in input read time with SVMLight format, and a 39.5% reduction in input read time with HDF5 format.

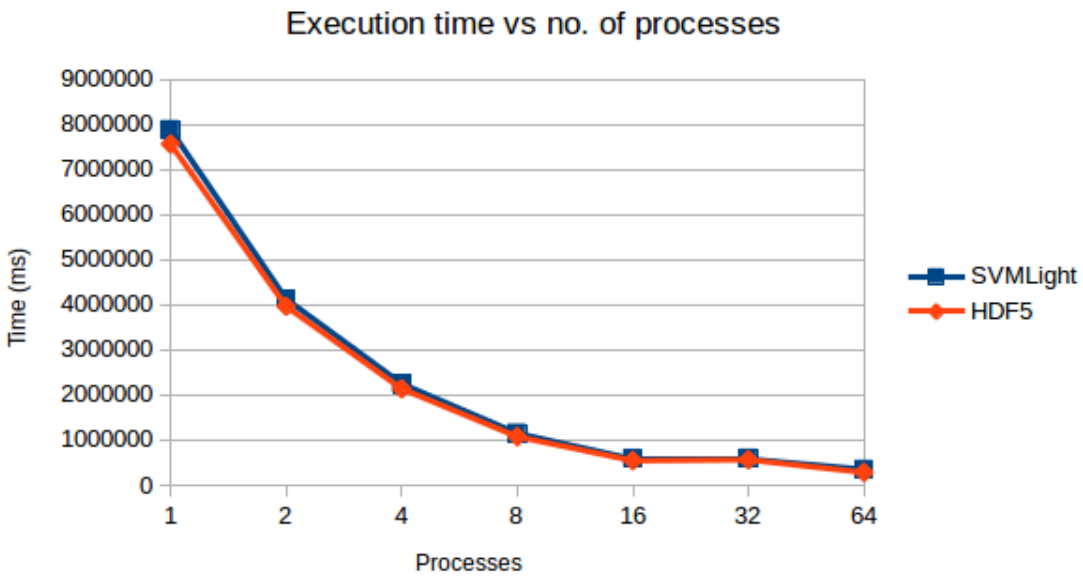

Figure 5.10: Total execution time compared between data formats as number of processes is increased. On average the reduction in execution time is 4.88% for 1 through 64 processes.

Figure 5.10 illustrates how big the processing part is in the total execution of PiSVM-Predict. Despite the improvements in input read speeds shown in figure 5.9 there is a modest reduction in the total execution speed between SVMLight and HDF5.

5.9 Reduction in file size

File sizes for Indian Pines Raw dataset.

File name	SVMLight size	HDF5 size in bytes	Percentage reduction
Train	83.014.311 bytes	22.631.709 bytes	72.74%
Test	747.135.597 bytes	197.528.589 bytes	73.56%
Model file	91.584.884 bytes	69.169.200 bytes	24.48%

Table 5.1: Reduction in file size

Table 5.1 illustrates an interesting case where the resulting model file is bigger than the input data file that was used for training. This is because almost all of the

vectors are chosen as SVs. The model file contains 32911 SVs whereas the data file contains 33000 vectors and so very few were discarded. Table 5.1 also illustrates that the greatest size reduction happens with data files. Data files contain two multi-dimensional datasets that benefit a lot from compression.

5.10 Changes in memory footprint

Memory footprint was not specifically measured as that was not considered the scope of this thesis. Memory footprint was however estimated based on the modifications done to the original code.

There are little changes to the memory footprint of PiSVM-Train during execution, the overhead of opening and reading a file is very small compared to storing the data from those file in memory. Both SVMLight and HDF5 files are read and the data is stored in identical structs in memory while it is processed.

There are more substantial changes to the memory footprint in PiSVM-Predict. When using SVMLight a line is read, processed and then the process moves onto the next line, discarding the first line from memory, so memory footprint is quite small. With HDF5, however, all the relevant lines are read into an array allocated on the heap, leading to a larger memory usage. This memory increase is thus relevant to the size of the dataset as it is all read into memory. This increased memory usage was not profiled, as that was considered outside the scope of the project, but rather kept to a minimum as much as possible by reading only the relevant lines and freeing the memory as soon as processing was done.

5.11 Summary

The evaluation was performed on JURECA, an HPC cluster at JSC with over 150,000 cores. The dataset used to measure improvements was the Indian Pines Raw dataset. It is both the biggest in terms of bytes and it is one of the more complex datasets available for this project, it has roughly 30000 vectors, 52 classes (originally 58 but 6 were discarded because of lack of features) and 200 features. It was gathered by the AVIRIS sensor over the Indian Pines test site in Indiana.

Time measurement was performed using C++ std::chrono high_resolution_clock. Timing measurements were performed on PiSVM-Parse to establish size reduction of files (greatest reduction of 73.6%) and parsing time with varying compression levels.

Based on the results shown in figures 5.1, 5.2, 5.3 and 5.4, the default compression level was set to 9 (highest) in PiSVM-Parse.

PiSVM-Train performance was measured with regards to compression levels. Uncompressed HDF5 read and write operations are the fastest but when compression is being used there is not a significant difference between compression levels, a 9.7% difference between the fastest and slowest input read time with any compression level (excluding level 0). It was thus decided to set the default compression level of PiSVM-Parse to 9. Output write time is improved by 98% by writing the model file in HDF5 format. Total execution time is reduced by 4.5% by using HDF5 data file compressed with level 9 compression.

PiSVM-Train performance was also measured with regards to increased process count. Input read times were reduced by 94.3% on average. Output read time was not improved as only the Rank 0 process writes out the model. Total execution time is reduced by 3.45% on average. An increase in measured times was observed between 16 and 32 processes at the 24 process mark. This increase can be explained by inter-node communication overhead.

PiSVM-Predict performance was measured with regards to increased process count. Input read time was reduced by 99.54% on average. No increase in measured times was observed at the 24 process mark as with PiSVM-Train as the classification operation does not need to communicate between processes.

File sizes were reduced. The data file size was reduced by 73% and the model file size was reduced by 24.48%.

Memory changes were not specifically recorded but design suggests there is no change in memory footprint of PiSVM-Train. Memory footprint increases in PiSVM-Predict as more memory is read into memory for classification.

6 Conclusion

Research questions answered:

1. In order to achieve speed increase in PiSVM, support for a complete framework and the associated file format had to be implemented.

2. The HDF5 library was chosen as it offered the greatest overall benefit, in terms of speed increase in I/O operations, in terms of flexibility of the dataset layout, and in terms of additional functionality it offered, like compression.

3. It was necessary to implement from scratch read and write functions for data into all pre-existing parts of PiSVM. PiSVM-Parse was added to the library, to parse data from SVMLight into HDF5 format. To PiSVM-Train functionality was added to read and write data into pre-existing structs that were then used during training. Parts in PiSVM-Predict had to be substantially rewritten in order to accommodate HDF5. No changes were made to PiSVM-Scale and at this point it does not offer an HDF5 support.

4. Changes in read and write speed were measured with high precision. In PiSVM-Train 94.3% reduction in input read time was achieved. 98% reduction in output write time was achieved. Since each processor of PiSVM-Train has to read all the data into memory it was not possible to implement any parallel reading scheme that could boost speed further. Training data is also generally smaller than test data, because of that the overall execution time was only reduced by 3.45%.

   In PiSVM-Predict a 99.54% reduction in read time was achieved, on average, as processor count goes up. This implementation utilizes chunking to increase read speed, and reduce the data needed to be read by each processor. The overall reduction in execution time was 4.88% on average.

5. Additional functionality that was implemented was compression and chunking. Compression enabled reduction of data files by about 72% and reduction of model files by about 24% without impacting read and write speeds. Chunking allowed parallel read of data while preserving memory.

6.1 Applications

The most notable application of this project is the speed increase in PiSVM. Other applications include using the code as a basis for adding HDF5 support to other HPC applications. As more and more application gain HDF5 support it is possible to move more and more data sets over to HDF5 format and thus reduce storage space as well.

6.2 Outlook

Some changes can be made to PiSVM-Parse to increase its functionality. It would be beneficial to add the functionality to convert SVMLight model files into HDF5 model files. MPI support could also be added so that parsing can be done in a truly parallel fashion.

PiSVM-Train could be updated to support parallel writing of the model file in HDF5 format. This however would require a more substantial rewrite of PiSVM-Train and the PSMO solver.

PiSVM-Scale needs to be updated to support reading, processing, and scaling of HDF5 files. No modifications were made to PiSVM-Scale in this thesis.

6.3 Source code

The source code of the PiSVM toolset with integrated HDF5 support is available at `https://gitlab.com/CaptainPants/pisvm`

It is distributed under the GNU General Public License v2.

Bibliography

[1] K. Þrastarson, "Design, implementation and analysis of a parallel and scalable cascade support vector machine framework," Master's thesis, University of Iceland, 2018. In press.

[2] R. A. Schowengerdt, *Remote Sensing: Models and Methods for Image Processing*. Elsevier, 2006.

[3] GrindGIS, "Remote sensing applications." `http://grindgis.com/remote-sensing/remote-sensing-applications`, 2016. [Online; accessed 18-02-2018].

[4] NASA, "Report of the eos/data panel." `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19860021622.pdf`, 1986. [Online; accessed 18-02-2018].

[5] B. Barney, "Introduction to parallel computing." `https://computing.llnl.gov/tutorials/parallel_comp/`, 2018. [Online; accessed 18-02-2018].

[6] G. M. Amdahl, "Computer architecture and amdahl's law," *IEEE Computer*, vol. 46, no. 12, pp. 38–46, 2013.

[7] W. L. Hosch, "Machine learning." `https://www.britannica.com/technology/machine-learning`, 2018. [Online; accessed 18-02-2018].

[8] A. L. Samuel, "Some studies in machine learning using the game of checkers." `http://ieeexplore.ieee.org/document/5392560/`, 1959. [Online; accessed 18-02-2018].

[9] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. MIT Press, 2012.

[10] N. Christianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, 2000.

[11] B. E. Boser, I. Guyon, and V. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992.* (D. Haussler, ed.), pp. 144–152, ACM, 1992.

[12] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[13] M. Sewell, "Support vector machines (svms) - history." `http://www.svms.org/history.html`. [Online; accessed 09-04-2018].

[14] S. Karamizadeh, S. M. Abdullah, M. Halimi, J. Shayan, and M. j. Rajabi, "Advantage and drawback of support vector machine functionality," in *2014 International Conference on Computer, Communications, and Control Technology (I4CT)*, pp. 63–65, Sept 2014.

[15] M. Haefele, "Parallel i/o for high performance computing." `http://calcul.math.cnrs.fr/Documents/Manifestations/CIRA2011/2011-01_haefele_parallel_IO-workshop_Lyon.pdf`. [Online; accessed 09-28-2017].

[16] D. E. Womble and D. S. Greenberg, "Parallel I/O: an introduction," *Parallel Computing*, vol. 23, no. 4-5, pp. 403–417, 1997.

[17] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters." `https://www.usenix.org/legacy/events/fast02/full_papers/schmuck/schmuck.pdf`. [Online; accessed 08-04-2018].

[18] PVFS, "Parallel virtual file system." `https://www.pvfs.org`. [Online; accessed 08-04-2018].

[19] Lustre, "Lustre." `http://lustre.org/`. [Online; accessed 08-04-2018].

[20] T. H. Group, "Introduction to hdf5." `https://support.hdfgroup.org/HDF5/doc/H5.intro.html`. [Online; accessed 07-10-2017].

[21] The HDF5 Group, "How is hdf5 different than hdf4?." `https://web.archive.org/web/20090330052722/http://www.hdfgroup.org/h5h4-diff.html`. [Online; accessed 08-04-2018].

[22] M. Richerzhagen, "Pisvm readme.txt." `https://github.com/mricherzhagen/pisvm/blob/cyclicDistribution/README.txt`. [Online; accessed 09-21-2017].

[23] beeblbrox, "Pisvm 1.2 homepage." `http://pisvm.sourceforge.net/`. [Online; accessed 09-21-2017].

[24] Various, "Projects supporting the hdf data format." `http://mloss.org/software/dataformat/hdf/`. [Online; accessed 21-04-2018].

[25] The HDF Group, "Hdf5 connector for apache spark (beta)." `https://www.hdfgroup.org/downloads/spark-connector`. [Online; accessed 21-04-2018].

[26] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade SVM," in *Advances in Neural Information Processing Systems 17 [Neural Information Processing Systems, NIPS 2004, December 13-18, 2004, Vancouver, British Columbia, Canada]*, pp. 521–528, 2004.

[27] E. Hartnett, "The netcdf tutorial." `https://www.unidata.ucar.edu/software/netcdf/netcdf-4/newdocs/netcdf-tutorial.pdf`. [Online; accessed 07-10-2017].

[28] CERN, "Root data analysis framework." `https://root.cern.ch/`. [Online; accessed 06-05-2018].

[29] A. Buckley, "The root of all evil." `http://insectnation.org/articles/problems-with-root.html`. [Online; accessed 06-05-2018].

[30] The Apache Software Foundation, "Apache spark." `https://spark.apache.org/`. [Online; accessed 08-04-2018].

[31] The Apache Software Foundation, "Apache hadoop." `https://hadoop.apache.org/`. [Online; accessed 08-04-2018].

[32] The Apache Software Foundation, "Apache spark mllib." `https://spark.apache.org/mllib/`. [Online; accessed 24-04-2018].

[33] The Apache Software Foundation, "Linear support vector machines (svms)." `https://spark.apache.org/docs/latest/mllib-linear-methods.html#linear-support-vector-machines-svms`. [Online; accessed 24-04-2018].

[34] R. Spitzer, "Apache spark cassandra connector." `https://github.com/datastax/spark-cassandra-connector`. [Online; accessed 06-06-2018].

[35] Z. Wen, J. Shi, B. He, Q. Li, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *To appear in arxiv*, 2018.

[36] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "Gpu acceleration for support vector machines," *Proc. 12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011)*, 2011.

[37] NVIDIA, "Cuda zone." `https://developer.nvidia.com/cuda-zone`. [Online; accessed 24-05-2018].

[38] HPS group @ JSC, Krause, Kondylis, "Jureca hardware and best practices." `https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/jurecapt16/jurecapt16-jureca.pdf`. [Online; accessed 20-05-2018].

[39] Top500.org, "Jureca." https://www.top500.org/system/178718. [Online; accessed 07-06-2018].

[40] M. Riedel, "Deep learning vs. traditional learning in remote sensing." http://www.morrisriedel.de/wp-content/uploads/2018/03/2018-03-22-Deep-Learning-vs-Traditional-Learning-Remote-Sensing-Riedel-v1.pdf. [Online; accessed 07-05-2018].

[41] Unknown, "Multispectral image: Rome, italy." https://b2share.eudat.eu/records/c203ec1815d14b84b162b5765f5d280f. [Online; accessed 07-05-2018].

[42] G. Cavallaro, M. Riedel, M. Richerzhagen, J. A. Benediktsson, and A. Plaza, "On understanding big data impacts in remotely sensed image classification using support vector machine methods," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, pp. 4634–4646, Oct 2015.

[43] S. P. Behrend, "Pisvm with hdf5 support source code." https://gitlab.com/CaptainPants/pisvm. [Online; accessed 04-07-2018].

[44] D. Zongker, "Chicken chicken chicken: Chicken chicken." https://isotropic.org/papers/chicken.pdf. [Online; accessed 08-06-2018].