Message Passing Interfaces and Support Vector Machines

Thursday 2nd January, 2025 - 17:07

Benjamin Vogel
University of Luxembourg
Email: benjamin.vogel.001@student.uni.lu

This report has been produced under the supervision of: Dr. Ezhilmathi Krishnasamy
Dr. Ezhilmathi Krishnasamy
University of Luxembourg
Email: ezhilmathi.krishnasamy@uni.lu

***Abstract*—This document aims to introduce the readers to the message-passing interface standard and its uses in distributed computing. Furthermore, it presents some fundamental topics regarding the semantics of that standard and gives examples of how to use MPI. Fundamentally, Point-To-Point and collective communication, as well as parallel I/O with MPI, are covered. Additionally, it explains how MPI is used to improve data-intensive programs as well as optimize the use of hardware for computational problems. Moreover, the second half of this document presents the SVM model for classification problems. The reader is provided with explanations on how SVMs are trained and how they make predictions, as well as how the program has been designed and implemented to make multi-classifications even though at its base, the SVM model is made of binary classification.**

1. Introduction

Data is an incredibly valuable resource that can provide a lot of valuable insight into many topics. As time goes on, data sets are only getting larger and larger such that a single computer does not possess sufficient computational power and memory to process the mountain of data that is being collected efficiently. Notably, in Machine learning, a great amount of data is being processed to train reliable models. These models are capable of making predictions, but to train these models, a lot of data has to be provided. Thus, the training data has to be distributed among multiple computing nodes so that the model can be trained in a timely fashion. The use of multiple machines connected to a network in order to solve a problem is called distributed computing. Distributed computing provides not only access to more computing power but also a larger amount of random-access memory that can be used for a given computational problem. Therefore, a standard for message-passing interfaces has been created to generalize how these compute nodes communicate with each other. This standard is presented within this document. Nevertheless, applying distributed computing within data science requires some understanding of the domain. Thus, this document also covers a section on the theory and implementation of SVMs and their use for the classification of data, along with the research regarding MPI.

2. Project description

2.1. Domains

2.1.1. Scientific. This project's scientific section centres itself within the domain of high performance computing and covers the Message Passing Interface standard and its usage for distributed computing.

2.1.2. Technical. This project's technical section is based on the domain of machine learning. Notably, the core of the technical deliverable is solving classification problems with SVMs. Moreover, the technical deliverable also touches on multivariable calculus when covering the training process of Support Vector Machines.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. A message-passing interface provides a programmer with a tool to communicate between processes of different compute nodes. But how does the usage of message-passing interfaces improve software performance? A good foundational knowledge of the basic structure of a Message Passing Interface, along with other knowledge regarding parallel computing, is necessary to answer such a question. This scientific deliverable provides the necessary knowledge on message-passing interfaces to understand when to use them, how message-passing interfaces are structured, and how MPI can be used to improve software performance.

2.2.2. Technical deliverables. The technical deliverable consists of a program capable of training SVM models using data provided by a user. The program simplifies the training

of SVM models and their use in making predictions on data.

Regarding Machine learning models, there are already many libraries that provide programmers with pre-built models. This is very convenient for programmers, as they can train and use these library-defined models without having to implement the entire model. However, it is usually better to have a proper grasp of what is being used instead of blindly working with libraries, not knowing what is happening behind the scenes. The technical deliverable consists of learning about support vector machines and building models from scratch. In fact, the program allows a user to train a model to make classifications using only the command line. Notably, the end goal for the technical deliverable is to be able to pass a dataset of characters to the program and receive a working character classification model.

2.3. Project main required competencies

2.3.1. Scientific main required competencies. It is recommended to already have experience with parallel computing, such as multi-threading, as well as experience with basic computer architecture and software scalability.

2.3.2. Technical main required competencies. For the technical section of this project, it is recommended that you have a good foundation in Python as well as the Numpy library. Finally, familiarity with multivariable calculus is advised.

3. MPI: How can Message Passing Interfaces be used to improve software performance?

3.1. Requirements

This semester project aims to discover how message-passing interfaces can be used to improve program performance and increase the scale of solvable computational problems. To elaborate, the scientific work is composed of multiple sub-sections. For one, the structure of a Message Passing Interface is presented in order to familiarize the reader with Message Passing Interfaces. The deliverable covers the most important topics and notions needed to have a sufficient understanding of how to create useful MPI programs. Optimally, the reader develops an intuition on when and how to use such an interface. Secondly, once familiar with the base structure of a message-passing interface, the scientific deliverable presents application cases for MPI. This includes MPI for I/O and matrix multiplication. In other words, this document is going to reference some MPI programs to better explain the interface usage.

3.2. Design

In order to provide the reader with the necessary knowledge of message-passing interfaces for this deliverable, the project first explains important concepts such as MPI operations and procedures, along with their different classifications. Additionally, Point-To-Point and Collective Communication, Communicators and Groups, and I/O are also explained. This knowledge will be taken from the official 4.0 documentation for the message passing interface standard. [8]. Exploring how message-passing interfaces are applied in various domains as well as how their impact is within scientific computing is extracted from these resources: [9] [5]. Moreover, the implementation of parallel matrix multiplication algorithms and I/O with MPI enables readers to better understand how the operations of message-passing interfaces can be used to solve problems. Finally, the project presents good practices for using Message Passing Interfaces, such as strategies to improve readability and debugging of MPI programs. Also, the deliverable informs the reader of issues that can occur due to improper usage of a Message Passing Interface. Good practices, as well as common mistakes, will be extracted from the MPI documentation [8] and a report from the Jülich Supercomputer Center [4]. As stated, example code as well as diagrams are provided to better visualize the concepts covered in this deliverable.

3.3. Production

Starting off, it is important to note that the MPI serves as a standard for message-passing interfaces. MPI is not a language or compiler specification, and it also does not represent a specific implementation.

3.3.1. Operations and Procedures. An MPI operation is defined as a sequence of tasks executed by a message-passing interface in order to establish a connection, transfer data, and synchronize processes. Operations usually take in various arguments providing information such as the source and destination processes for communications, the type and size of the data being transported, and other details covered later in this document. An Operation can be divided into 4 stages:

1. The Initialization procedure provides the operation with all arguments given except for the data buffer. Until the Freeing procedure is called, these arguments should be untouched by the user.
2. The Starting Procedure hands over control of the data buffer to the operation, which also shouldn't be accessed by the user until the completion procedure has finished.
3. The Completion procedure informs the program that buffers and arguments have been modified as well as allowing the user to access the data buffer again.
4. The Freeing procedure permits the user to access and utilize the arguments again and concludes the Operation.

Among Operations, we distinguish between 3 different types: Blocking operations, non-blocking operations, and

persistent operations:

Blocking operations combine all 4 stages into a single call. In other words, once the operation is initiated, control is only returned to the calling function once the operation has been freed.

Non-blocking operations are structured into two separate procedures. First, the initialization and starting stages are combined into a procedure, whereas the completion and freeing stages are combined into another procedure. As a result, the calling function gains control once the operation as been initiated. Nonetheless, this causes the needed to manually check whether the operation has completed, as there will not be any guarantee of completion from a single call to a non-blocking operation.

Persistent operations have a procedure for each of the stages. This allows the operation to keep a connection between two processes open and thus decreases overhead time when two processes repeatedely communicate with eachother.

MPI operations may be local or non-local. If an operation requires another process to call an MPI operation to return control to the program, the procedure is considered non-local. Otherwise, we call an operation local.

3.3.2. MPI Basic DataTypes. MPI defines its own basic datatypes. These data types have to be used by the sending and receiving operations to indicate what kind of data is sent/received. Using these pre-defined MPI datatypes, it is guaranteed that, even if data is sent between systems with different memory organizations, the data is sent and received without being corrupted.

3.3.3. Tags. Tags are used by processes to give extra context to the messages being sent. This is useful when a process sends two different messages to another process. Tags may be used to clarify the destination in order to make sure that each message is received by the correct receive operation.

3.3.4. Groups, Contexts, and Communicators. Communicators define the ”space” in which processes communicate with each other. This is done through groups and contexts tied to the communicator. Groups are ordered sets containing process identifiers, also called the rank of the process. These identifiers are integers ranging from 0 to ’the number of processes in the communicator’-1 and are used to reference processes when communicating. On the other hand, contexts are used to partition the communication space. In other words, Messages from one context cannot be received within another. This is useful when communications among specified sets of processes are supposed to be separated. A message-passing interface defines a default global communicator such as MPI_COMM_WORLD that contains all processes initially present on program execution.

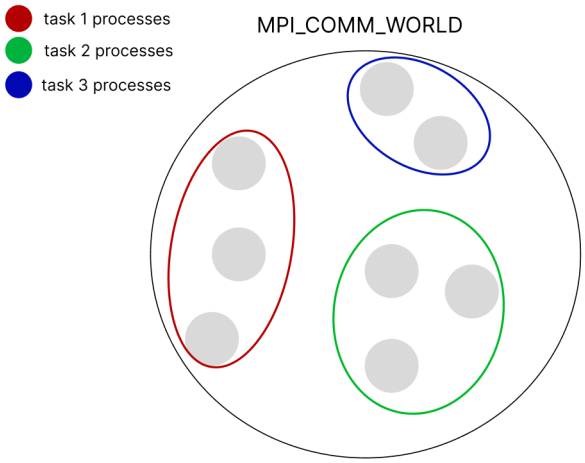

Figure 1: MPI_COMM_WORLD and how it can be subdivided into further comms

3.3.5. Point-to-Point Communication. MPI_Send and MPI_Recv are the fundamental operations within a message-passing interface. Nonetheless, before covering these operations, it is important to know how to set up an MPI environment. The operators MPI_Init, MPI_Comm_size, MPI_Comm_Rank, and MPI_Finalize are essential to any MPI program. MPI_Init initializes the MPI execution environment and takes the command line arguments (argv and argc) as input. MPI_Comm_Size and MPI_Comm_Rank take a communicator and an address to an integer variable, storing the size of the communicator and the rank of the process within that communicator, respectively. Finally, MPI_Finalize terminates the MPI environment and should be called before returning from execution. Regarding thread safety, the thread calling the MPI_Init function will be considered the main thread and is thus also the thread that should call MPI_Finalize.

Fundamentally, MPI_Send and MPI_Recv are enough to allow processes to communicate with each other. Of course, many other operations make certain tasks a lot easier to accomplish. Nonetheless, this document begins by covering these two basic operations.

MPI communications work with buffers to send and receive data. The sending operation takes the message’s content and either passes it to the system buffer, where the message content waits to be received, or it immediately sends the message to the receiving process if it is already awaiting the message. How this is handled can vary from different MPI implementations. For a communication to go through, it is necessary to provide the sending process with information on the message content along with something called an envelope. First, the message content is composed of the content’s location in memory, the number of elements contained at that location, and the data type of the elements. Secondly, the message envelope contains information that helps MPI to distinguish messages and

coordinate them correctly between processes. The message envelope contains the rank of the sending and receiving process, the tag of the message, and the communicator. Similarly, the receiving operation also requires a location where to store the message content, the **maximum amount** of data expected to be received, the type of the message content, as well as the expected fields of the message envelope. In other words, the rank of the sending process, the tag set by the sending process along with a common communicator are to be provided. Important to note is that the send operation presented is in standard communication mode. Other communication modes exist, such as buffered mode, synchronous mode, and ready mode.

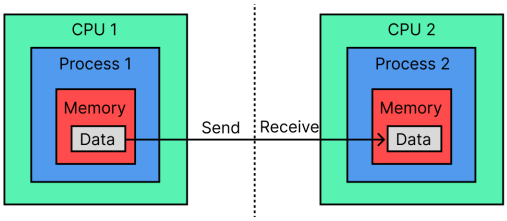

Figure 2: Point-To-Point Communication

The standard communication mode send is a blocking operation that takes a message and an envelope and only returns once the data buffer becomes available again, which often is tied to requiring a matching receive. Thus, the standard communication mode is non-local. The buffered mode uses a separate user-provided buffer to store messages. This way a program can continue its execution without having to wait for the message to be copied to the system buffer or having a matching receive posted, because the message gets saved within the user buffer until the message can be copied into the system buffer or immediately sent to the destination process. Thus, the operation is local, as its completion does not depend on the call of a matching received from a different process. Note that the management of this buffer is up to the user, and if there is insufficient buffer space, an error will occur. Next, the synchronous mode can also be posted whether or not a matching receive has been posted. However, the send will only be completed once a matching receive is posted and the receive operator has started to receive the message. The completion of a synchronous call indicates not only that the buffer can be reused but also that the receiver is executing the matching received. As a result, a synchronous communication call is non-local. Finally, the ready communication mode only starts once there is already a matching receive posted. If this is not the case, there will be an error and the outcome is undefined. This communication mode allows for the removal of a handshake protocol, which improves performance. Despite having various communication modes, there is only one receive operation, which is capable of receiving messages from all sending operations. The receive operation is only completed once the receive buffer contains the message data.

Communication mode	Function	Locality
Standard	MPI_Send	non-local
Buffered	MPI_Bsend	local
Synchronous	MPI_Ssend	non-local
Ready	MPI_Rsend	non-local

TABLE 1: Communication mode characteristics (blocking)

All these operations are blocking operations as each operation, on call, blocks the execution of the calling function until a certain condition has been fulfilled. This can cause deadlocks within the MPI application if processes are waiting for sends or receives that are never posted. However, such deadlocks can be avoided through the use of communication patterns, the MPI_SENDRECV operation, which posts a send and receive simultaneously or the use of non-blocking operations [4]. To elaborate, MPI provides a set of non-blocking sending operations containing the same communication modes as seen previously. These operations immediately return control to the calling function after the send operation has started and are completed using a separate send-complete call such as MPI_Wait and MPI_Test. Thus, it is the responsibility of the user to verify whether a non-blocking send or receive operation has been completed. Consequently, because the execution of a function is not blocked by the operation calls, the program is capable of performing multiple communications concurrently and can work on other tasks while communications are being processed. This allows for more complex programs and can improve program performance.

MPI_Wait and MPI_Test are used to complete non-blocking communications. The wait function blocks the execution of the program until the communication is complete. MPI_Test checks for the completion of the communication but does not hinder the execution of a program. The completion of a send operation indicates that the send buffer is available again for use, and the completion of a receive operation indicates that the receive buffer contains the message content.

Finally, if two processes repeatedly communicate with each other, it may be inefficient to establish a connection between the processes over and over. Thus, MPI provides a programmer with persistent communication requests, which allow processes to establish a connection and maintain it for a certain amount of time, avoiding the initialization overhead.

3.3.6. MPI Collective Communication. Point-to-point communication is the foundation of a message-passing interface and can be used to create very fine-grained communication setups.

Nonetheless, the communication setups of many MPI

programs are similar. In short, many programs want to distribute data equally among processes and gather the newly produced information afterwards.

Conveniently, MPI provides operations that facilitate the implementation of that kind of data transfer. These operations are grouped under the term Collective Communication. Note that collective communications can be classified as One-to-All, All-to-One, and All-to-All. Moreover, it is necessary to make sure that all processes within the concerned communicator call these collective operations. Otherwise, the program is going to have undefined behaviour as there are going to be idle processes waiting for all processes within the communicator to make the operation call. [4]

MPI_BCAST and MPI_SCATTER are One-To-All operations. MPI_BCAST takes the data provided by the sending process and broadcasts the data to all other processes. As a result, every process receives the same copy of the sender's data. Similarly, MPI_SCATTER receives data from the sender and distributes it equally among all other processes. After using MPI_SCATTER, each process has a different part of the same size of the message content.

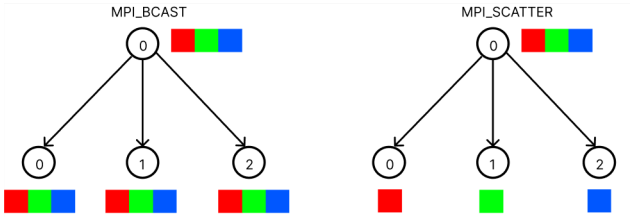

Figure 3: Data distribution - broadcasting and scattering

MPI_GATHER and MPI_REDUCE are All-to-One operations and are frequently used in tandem with the One-To-All operations presented previously. MPI_GATHER does the opposite of MPI_SCATTER, taking the data from all processes and sending the collection to a single process. If the program wants to perform operations on the collection of information, it can use MPI_REDUCE to take the collection of data, operate on it, and send the result to the receiving process. MPI has a set of predefined operations, such as MPI_SUM, that can be used by the MPI_Reduce operation. Additionally, the programmer can also create their own MPI operations using the MPI_OP_CREATE operation. Summing up the results made by all processes and sending it to the root process is an example where MPI_REDUCE is appropriate to use.
Finally, the All-To-All operation MPI_ALLGATHER collects the information from the sending buffer of all processes, but instead of sending it to only a single process, it sends the information to all processes.

3.3.7. MPI I/O. Parallel I/O is very difficult to implement. Thus, MPI provides an interface to facilitate operations on

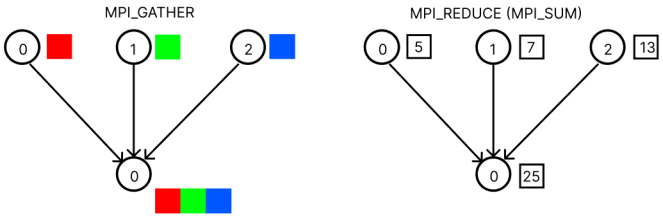

Figure 4: Data collection - gathering and reduction

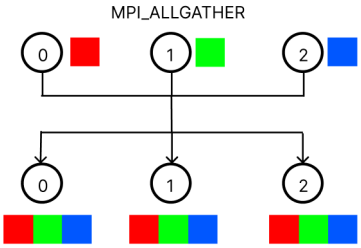

Figure 5: Data collection - ALLGATHER

files within a parallel environment.

MPI has its own definitions regarding I/O [8]:

A file is considered an ordered collection of typed data elements. A file is opened using the MPI_FILE_OPEN operations. The function requires to specify a communicator, i.e. the group of processes that are going to access the file, the name of the file that is to be opened, the mode in which the file is to be opened, an info object, and a variable in which the file handle is going to be stored. The file handle is used by file operations to reference the file. Message-passing interfaces provide 3 different ways to access data from a file: explicit offsets, individual file pointers, and shared file pointers. With explicit offsets, there are no file pointers, and the position within the file from where a process is supposed to read or write is explicitly provided to the operation. MPI_FILE_(READ/WRITE)_AT is linked to data access with explicit offsets. File pointers differ compared to explicit offsets because, instead of having to provide the offset explicitly, MPI maintains a file pointer for each process for each file handler. Thus, instead of explicitly giving the offset manually for an operation, MPI tracks where the next operation should read or write. MPI_FILE_(READ/WRITE) are associated with individual file pointer data access. Finally, shared file pointers are, as the name implies, file pointers shared by processes. The offset is again stored and updated. However, when all processes call a collective file operation, the offset is determined using the shared file pointer and the rank of the processes. MPI_FILE_(READ/WRITE)_SHARED is associated with shared file pointers. Important to note is that when multiple processes try to use file operations that use shared file pointers at the same time, these operations are defined to behave like serial calls, meaning that in collective

calls, the process with the lowest rank accesses the earliest part of the file. Furthermore, when non collective shared file pointer operations are used, it is not guaranteed that the file access will be in the order of the the file rank.

Just like with point-to-point communication, each file operation has its collective and non-blocking counterpart. The collective operations add the term _ALL (MP_FILE_WRITE_ALL) at the end of the original operation, whereas the non-blocking operations append and I in front for the "READ/WRITE" (MPI_FILE_IWRITE).

MPI applications I/O can become quite fast if collective I/O and predefined file views are taken advantage of. The file view defines the current set of data visible and accessible from an open file. More specifically, the view of a file is comprised of a displacement indicating how many bytes are skipped starting from the head of the file when accessing the data, the type, which is just the datatype/data structure read or written, and the file type. The file type specifies what part of the file is visible by the process at all times. [10] Providing sufficient information to the message-passing interface allows the compiler to optimize file access.

3.3.8. MPI General Datatypes. With the predefined basic datatypes provided by MPI, the programmer is capable of sending and receiving a set of a single data type between processes. However, a programmer is not capable of passing messages of structures, nor can he pass non-contiguous data. Therefore, MPI provides functions to create more general MPI-datatypes.

A general datatype consists of two things:

- a sequence of basic datatypes
- a sequence of integer (byte) displacements

The new datatypes sequence of basic-datatype/displacement pairs are called its type map, and the sequence of basic-datatypes is called its signature:

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

$$Typesig = \{type_0, \ldots, type_{n-1}\}$$

Furthermore, the extent of a datatype is the number of bytes needed to store the datatype such that it is aligned in memory. According to [8] if:

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

then

$$lb(Typemap) = \min_j disp_j$$
$$ub(Typemap) = \max_j (disp_j + sizeof(type_j)) + \epsilon$$
$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

where ϵ is the padding needed to align the datatype in memory.

Knowing this, it is now possible to create new general datatypes using functions such as MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, and MPI_TYPE_STRUCT. These can be used to make more complex communications or very specific accesses to files.

As an example, creating a datatype with MPI_TYPE_VECTOR would go as follows: The MPI_TYPE_VECTOR function requires 5 arguments. For one, it requires a type map, also called an old type, that is going to be used for the new data type, as well as a handle that is going to be used to reference the newly created data type. Next, it is necessary to indicate the number of blocks contained in the data type as well as their length. The block length indicates how many times the given type map gets repeated per block. Finally, the function requests a stride, which indicates the number of bytes between the beginning of each block. The stride gets calculated by multiplying the stride given as an argument by the extent of the old datatype. Given:

$$oldType = \{(MPI\_DOUBLE, 0), (MPI\_CHAR, 8)\}$$
$$extent(oldType) = 16$$

Executing MPI_TYPE_VECTOR(3,2,3,oldType,newType), the following datatype is going to be referenced by newType:

$$\{(MPI\_DOUBLE, 0), (MPI\_CHAR, 8),$$
$$(MPI\_DOUBLE, 48), (MPI\_CHAR, 56),$$
$$(MPI\_DOUBLE, 96), (MPI\_CHAR, 104)\}$$

After having created a new datatype, it is necessary to commit it before the datatype can be used in a communication. MPI_TYPE_COMMIT is used therefore. Once the data type is not needed anymore, MPI_TYPE_FREE marks the datatype for deallocation and sets the datatype to MPI_DATATYPE_NULL. Note that all still active communications with that datatype are completed normally. Finally, freeing a datatype does not affect other datatypes constructed from it.

3.3.9. Distributed model vs Shared memory model. Message-passing interfaces provide a well-structured and versatile interface for distributed computing. One might ask oneself, however, how distributed computing differs from the shared memory model. The distributed model and the shared memory model for parallel computing have many differences and should be seen as two tools for different tasks. On the one hand, the shared memory model allows for multiprocessor machines to accelerate computations by sharing the memory space and operating in parallel [13]. Notice that for the shared memory model, communication between processes or threads is fast, but it is important to pay attention when accessing memory due to possible race conditions [1]. On the other hand, distributed computing provides more than just additional computing power because

it also allows the use of further memory from other compute nodes, providing a solution to data-intensive applications. Nonetheless, communications between processes become a lot more costly compared to their shared memory counterpart and thus have to be used wisely.

3.3.10. Example Programs. The two programs referenced in this section can be found here:

Matrix Multiplication using MPI
First, let us analyse a possible implementation of a matrix multiplication function using MPI. This section is going to cover the communication structure of the matrix_mult function. The idea behind this algorithm is to assign each processor a row belonging to the resulting matrix. To elaborate, each process has a row from the first matrix assigned to it and receives another row from the second matrix. The processes do the appropriate computations with the two-row matrices and store the results in a 3rd array designated to store a row belonging to the final matrix. After finishing the computations, the processes pass around the rows of the second matrix such that at the end of the algorithm, each process receives every row of the second matrix exactly once. As a result, each processor should have finished computing the row belonging to the resulting matrix. Afterwards, the rows are gathered by the root processes, and the matrix is returned.

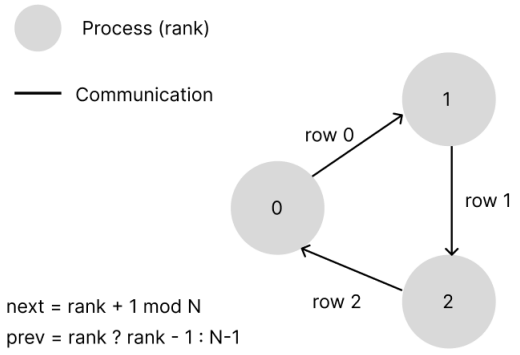

Figure 6: Passing of matrix rows among processes.

N processes are used for a matrix multiplication between two square matrices A and B of size n. Notably, for process i, which computes the ith row of the output matrix C, being assigned the ith row of A and currently possessing the kth row of B:

$$c_{ij} = c_{ij} + a_{ik}b_{kj}$$

repeating this for all n-1 rotations gives us the exact same result as the original formula for the element of the resulting matrix due to matrix multiplication:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}$$

Reading and Writing to Files using MPI

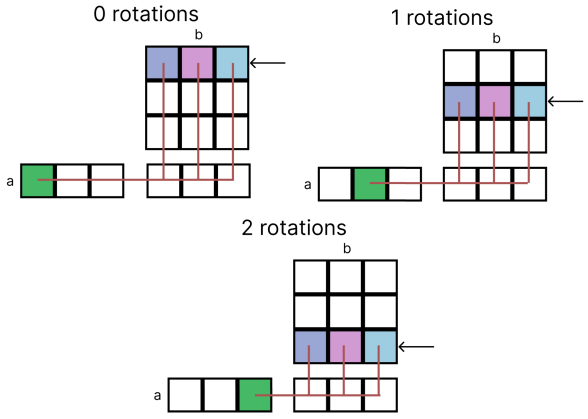

Figure 7: MPI matrix multiplication

The next program consists of reading two matrices from a binary file, computing the matrix addition, and saving the result in another file.

Simply put, the program opens both files containing the matrices as well as the third file that is going to store the resulting matrix. Then, the program reads the matrix entries, sums the matrix entries element-wise, and then writes them into the output file using the explicit offset method.

The offset for file access is computed using the rank of each processor along with the total number of processes, the number of elements read by each processor at once, and the size of the datatype read in bytes:

$$offset = rank + (i * size) * nints * sizeof(int)$$

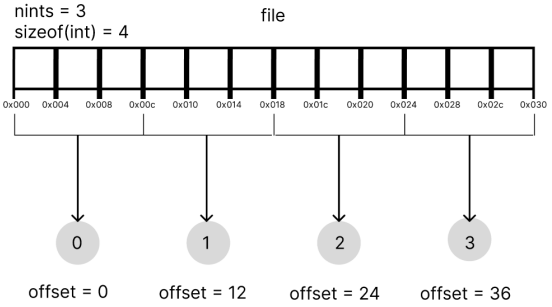

Figure 8: How offset directs file access

A possible improvement to this implementation would be to utilize collective file operations such as MPI_FILE_READ_ALL alongside the use of file views.

More code can be found at: github-BSP-S3

3.3.11. Advice regarding the use of MPI. First and foremost, it is important to understand that even though

we have access to a message-passing interface, one should make sure to keep communications to a minimum. Communications take extra time and memory and should not be seen as a tool without any costs. That is why it is recommended to combine messages as well as use collective communications wherever possible, as it improves buffer use and decreases communication latency and overhead. [4]

Furthermore, regarding Point-to-Point communication, even though synchronous communication conserves resources by not requiring an extra buffer, it should be used thoughtfully, as synchronous communications can lead to the serialization of a program and cause processes to remain idle for an extended amount of time. Generally speaking, it is safe to use the standard MPI_SEND operation because it allows the message-passing interface to optimize communications. [4]

3.3.12. MPI as a tool. Message-passing interfaces are very important regarding high-performance computing, as they allow for processes to communicate and coordinate. This frees us from the constraint of scaling solely based on the individual processor capabilities. Message-passing interfaces connect compute nodes of an HPC cluster, for example, and thus allow programs to make use of multiple CPUs, GPUs, and a lot more memory. The additional random access memory provided through the use of multiple compute nodes is especially useful in the context of data analysis and machine learning, where it becomes more and more difficult to process the mountains of data that are being collected daily [9]. Furthermore, message-passing interfaces provide tools to further increase the performance of a single compute node by possibly coordinating different processes, which each give instructions to the CPU and GPU, respectively. This maximizes their usage, making sure as little resources as possible are wasted. Again, maximizing the usage of compute node resources and optimizing data reading and writing are essential in domains such as data science and simulations. [9] [5].

3.4. Assessment

The aim of the Scientific deliverable of this project is to present the general structure of Message Passing Interfaces, provide code examples and give example scenarios where MPIs could be useful. First, this document provides an introduction to the basic concepts within message-passing interfaces like the basic datatypes, communicators, point-to-point communication and collective communication. Additionally, parallel I/O with MPI, as well as General Datatypes, have also been covered, giving the reader already enough information to create useful MPI programs. Furthermore, the scientific deliverable presents two example programs: a matrix multiplication program used to present the usage of collective and point-to-point communication, as well as an I/O program used to present the usage and efficiency of parallel I/O using MPI. Finally, the document covered interesting and relevant scenarios where message-passing interfaces could be used. All in all, the scientific deliverable has been a success.

4. SVM: Character Recognition

4.1. Requirements

The program can be partitioned into 3 sub-components:

- The SVM model
- The component in charge of managing the multiple SVM models
- The component in charge of processing the command line arguments

The SVM model can be trained to make classifications on data. Regarding data, the SVM is only capable of receiving a dataset, which has been processed to only contain {+1,-1} labels. In short, the SVM model can be trained and make predictions. The managing component is tasked with the coordination of the SVM models as well as processing and providing the SVM models with usable data. The model managing unit serves as an interface for the usage of the SVM models. In other words, the SVM model is only operated by the model manager.

The managing component is tasked with the distribution of data among multiple SVM models as well as their collective training. Additionally, the manager coordinates the models to make predictions with the provided data. Parameters like data sets, for example, are passed to the managing component by the argument processing component. Similar to the relationship between the SVM model and the model manager, the managing component is directly controlled by the processing unit.

Finally, the argument processing component takes the command-line arguments given by the users and delegates the appropriate tasks to the model managing component based on the arguments. Furthermore, it is the argument processing component's job to communicate with the user and to inform the user of results or possible errors raised by the model managing component.

Combining everything, the program will be utilizing a Support Vector Machine model to make classifications. Because an SVM makes binary classifications, multiple SVM models are trained to allow multi-classifications. The number of models is proportional to the number of different labels provided by a dataset. These SVMs are managed by a model manager who is in charge of distributing and training the models as well as of the classification of data using the models. The user can provide the program with a dataset along with instructions which are analyzed by the argument processing component and passed to the model managing component. The model manager can train the SVM models, storing the determined weights for the

classification model in a text file. Additionally, the user can request to make predictions, where the model manager loads the SVM models into memory and performs the classification.

4.2. Design

The Support Vector Machine model is the building block of this deliverable. Hence, first and foremost, the SVM model is described, and the definition of its training process is determined and implemented [3]. Notably, for this Support Vector Machine model, John C. Platts Sequential Minimal Optimization algorithm is used for the training process [12]. The SVM model contains all the parameters necessary for the training of the model and the classification of the data. Additionally, the model has a method for setting the training data and the hyperparameters, respectively. Moreover, methods for the training of the model, along with a prediction method, are needed. Lastly, getters and setters for the support vector machine weights are needed to store and load the SVM weights.

The model managing component has to perform the following tasks: receive data sets, appropriately modify data for the SVM training process, manage the training of the SVMs by delegating the models and storing the resulting weights in files as well as load the weights of a model from a file into memory and coordinating the SVMs when classifying data.

The model manager can be seen as a tool for the processing component to streamline the usage of the SVM models. It is the argument processing components' job to provide the data to the model manager and inform it whether to train a model or make classifications using an already trained model. For the training process, the appropriate number of SVMs are instantiated and provided with the data passed by the user. As previously stated, each model is responsible for a $\{y$, not-$y\}$ classification, where y is one of the possible labels that can be assigned to a data point. Once these models have accomplished their training, the resulting weights determined by the SVMs are stored in a file. On the other hand, for the classification of data, the model manager loads the information of the trained SVM models into memory. The results of the classifications are printed in a file.

Finally, the argument-processing component is in charge of the communication between the user and the model-managing component. The user can provide various information, such as datasets and indicators for whether a new model should be trained or new classifications are to be made. Additionally, the user can also provide the name under which a model's information is to be stored. It is the argument processor's job to make sure that the appropriate arguments have been passed and, if so, how they are interpreted. Additionally, the argument processing component is also tasked with communicating with the user any information, including prediction or training results or any errors that may have occurred. The processing component should receive and appropriately handle any errors that have been raised by the managing component.

Concerning program libraries and APIs, only the Python standard library, as well as the NumPy library, are to be used to produce this program. Nonetheless, already available APIs such as sklearn are used to benchmark the technical deliverables regarding accuracy and robustness. This provides the technical deliverable with a method to evaluate the final product as well as to present the difference in performance between the implementations.

4.3. Production

The production section of the technical deliverable presents how the software has been implemented, which includes explaining the concept of Support Vector Machines and providing further explanations on how an SVM can be produced.

Support Vector Machines are a machine learning model mainly used for classification problems. Fundamentally, a Support Vector machine takes a dataset containing data representing two different classes, usually labelled {+1,-1}, and searches for a hyperplane that is capable of separating the two data classes optimally. The idea is that once a suitable hyperplane is found, the model can classify new data based on the position of the data point with respect to the found hyperplane. This hyperplane is called the Decision Boundary.

An optimal hyperplane is a hyperplane whose margin is maximal. The margin is defined as the distance between the hyperplane and the data points closest to that hyperplane. These data points are also called support vectors.

In its most basic form, SVMs can only classify linearly separable data. However, more sophisticated implementations of SVMs can classify almost linearly separable data as well as non-linearly separable data.

This section starts by explaining how the simplest form of SVMs, the Hard-Margin SVM, works. Afterwards, the project builds upon that knowledge until a usable SVM model is produced:

Hard-Margin SVM
Assume a set of linearly separable data points (x,y), where x is the input features and y is its label. Because an SVM specializes in binary classification, $y \in \{-1, 1\}$.

Let the decision boundary be given by

$$DB := \mathbf{w} \cdot \mathbf{x} - b = 0 \tag{1}$$

where **w** is a vector containing the model weights, and b is the intercept of the hyperplane. Some attributes of **w** are that it has the same size as the input feature vectors and

that it is a normal vector of the decision boundary.

What's more, Margin lines are lines parallel to the Decision Boundary. They are placed such that they pass through the Support Vectors of the Dataset. These lines are given by:

$$ML := \mathbf{w} \cdot \mathbf{x} - b = \pm 1 \tag{2}$$

The distance between the two margin lines defines the mar-

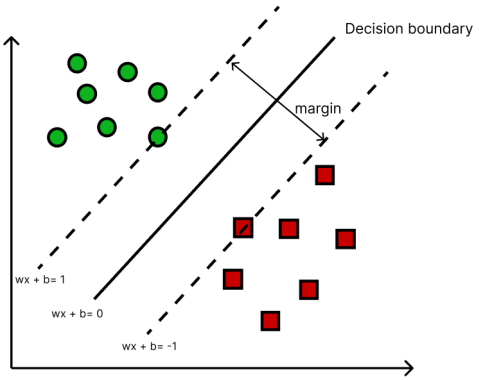

Figure 9: Decision boundary and Margin Lines visualized

gin. Usually, the larger the margin, the better the predictions. However, how is the margin maximized for a given dataset? First, let us determine the exact expression for the margin. Starting with the distance between the decision boundary and a margin line, for any point on the decision boundary, a point has to move k number of steps into the direction of $\mathbf{w}$ to reach the margin line. If we determine k, we know the distance between the DB and the Margin line:

$$\mathbf{w} \cdot (\mathbf{x} + k\frac{\mathbf{w}}{||\mathbf{w}||}) - b = 1 \tag{3}$$

Developing the equation, we get:

$$\mathbf{w} \cdot \mathbf{x} + k\frac{\mathbf{w} \cdot \mathbf{w}}{||\mathbf{w}||} - b = 1 \tag{4}$$

By (1):

$$k\frac{\mathbf{w} \cdot \mathbf{w}}{||\mathbf{w}||} = 1 \tag{5}$$

This becomes finally:

$$k = \frac{1}{||\mathbf{w}||} \tag{6}$$

The actual margin is equal to twice that distance k:

$$\text{margin} = \frac{2}{||\mathbf{w}||} \tag{7}$$

Thus, maximizing the margin is equal to maximizing $\frac{2}{||\mathbf{w}||}$. This can be rewritten as a minimization problem by inverting the numerator and denominator. Nonetheless, this problem possesses the constraint that no data point is allowed within the margin region, and misclassification is not tolerated at all. Described as an optimization problem, taking $||w||^2$ as the objective function, instead of $||w||$, in order to facilitate computations later on:

$$\begin{aligned} \min_{w,b} \quad & \frac{||w||^2}{2} \\ \text{s.t.} \quad & y_i(w \cdot x_i - b) - 1 \geq 0 \\ & \forall i = 1, ..., N \end{aligned}$$

Note that the data point $\mathbf{x_i}$ where $y_i(wx_i - b) \pm 1 = 0$ are the support vectors and thus determine the positioning of the margin line.

Because the solution of the Hard-Margin optimization problem is almost identical to the Soft-Margin case, the explanation of solving this problem is postponed.

The hard-margin model allows us to classify linearly separable data. However, often, data is not this perfect. A single, unfortunately, placed data point could ruin the quality of the model's performance. Additionally, there may be outliers in the data that would make it impossible for the Hard-margin SVM to linearly separate the data:

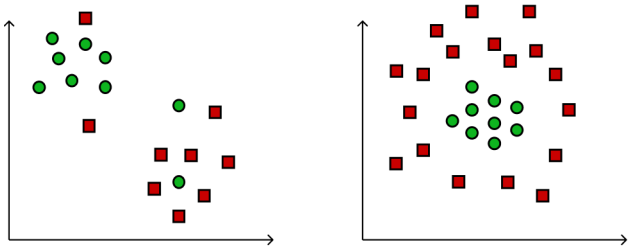

Figure 10: Datasets that are non-suitable for Hard-Margin SVMs

Therefore, it is necessary to adapt the model to take into account misclassifications and adjust the weights accordingly without breaking the model entirely.

Soft-Margin SVM

In order to handle data that is not entirely linearly separable, we introduce a new variable ξ such that:

$$y_i(w \cdot x_i) - b \geq 1 - \xi_i \quad s.t. \quad \xi_i \geq 0 \tag{8}$$

ξ provides some tolerance for the classification of data points. Modifying our model also modifies the optimization problem that has to be solved during the training process:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{||w||^2}{2} + C\sum_{i=1}^{N} \xi_i \\ \text{s.t.} \quad & y_i(w \cdot x_i - b) - 1 + \xi \geq 0 \\ & \forall i = 1, ..., N \end{aligned}$$

The farther away data points are from their expected region, the larger the penalty on the objective function. The hyperparameter C defines the trade-off between tolerance for the classification and the margin size. The larger the C is, the bigger the tolerance will be.

In order to solve this constrained optimization problem, the Lagrangian is established, with $\xi_i, v_i \geq 0$: [3]

$$L = \frac{||w||^2}{2} + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\lambda_i[y_i(w \cdot x_i - b) - 1 + \xi_i] - \sum_{i=1}^{N} v_i\xi_i \tag{9}$$

Now take the partial derivative of the objective functions parameters.

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{N}\lambda_i y_i x_i \tag{10}$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{N}\lambda_i y_i \tag{11}$$

$$\frac{\partial L}{\partial \xi_i} = C - \lambda_i - v_i\xi_i \tag{12}$$

Setting these partial derivatives to 0 gives us:

$$w = \sum_{i=1}^{N}\lambda_i y_i x_i \tag{13}$$

$$\sum_{i=1}^{N}\lambda_i y_i = 0 \tag{14}$$

$$C = \lambda_i + v_i \tag{15}$$

Substituting (13),(14) and (15) into (9) provides (7.1.1):

$$L = \sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i x_j \tag{16}$$

Note that this formulation is only in terms of the Lagrange multipliers. In fact, using this equation, it is possible to define the Dual of the optimization problem, with the Dual being defined as:

$$\max_{\lambda} \inf_{w,b,\xi} L(x, w, \xi)$$

Because it was possible to eliminate all w,b,ξ from (9), we ended up with the following optimization problem:

$$\max_{\lambda} \quad \sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i x_j$$
$$\text{s.t.} \quad \lambda_i \geq 0\ ,\ \forall_i \quad \sum_{i=1}^{N}\lambda_i y_i = 0$$

The solution to the Dual Problem is equivalent to the solution of the original problem as long as the solutions satisfy the KKT conditions:

$$\begin{cases} \lambda_i = 0 \implies y_i(w \cdot x_i + b) \geq 1 - \epsilon \\ \lambda_i = C \implies y_i(w_i + b) \leq 1 + \epsilon \\ 0 < \lambda_i < C \implies y_i(w_i + b) \in (1 - \epsilon, 1 + \epsilon) \end{cases} \tag{17}$$

With epsilon being a tolerance for error. In order to solve this problem using the sequential minimal optimization, we first multiply (16) by -1 to turn the maximization problem into a minimization problem.

$$\min_{\lambda} \quad \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i x_j - \sum_{i=1}^{N}\lambda_i$$
$$\text{s.t.} \quad \lambda_i \geq 0\ ,\ \forall_i \quad \sum_{i=1}^{N}\lambda_i y_i = 0$$

With this new formulation of the SVM optimization problem, we can now utilize the SMO algorithm in order to find the set of λ_i that minimizes the function L(λ).

Sequential Minimal Optimization

This section covers the theory and implementation of the SMO algorithm for SVM training.

Simply put, the SMO algorithm performs coordinate descent. In other words, in each iteration of the algorithm, two random Lagrange multipliers are picked, and the function is minimized as if all other multipliers were constant values. It is not possible to minimize the multipliers one at a time, as the condition $\sum_{i=1}^{N}\lambda_i y_i = 0$ would not hold. It also follows that: [6]

$$y_1\lambda_1^{old} + y_2\lambda_2^{old} = y_1\lambda_1^{optimal} + y_2\lambda_2^{optimal} \tag{18}$$

Graphically speaking, the point (λ_1, λ_2) is contained in a box as $0 \geq \lambda_1, \lambda_2 \geq C$:

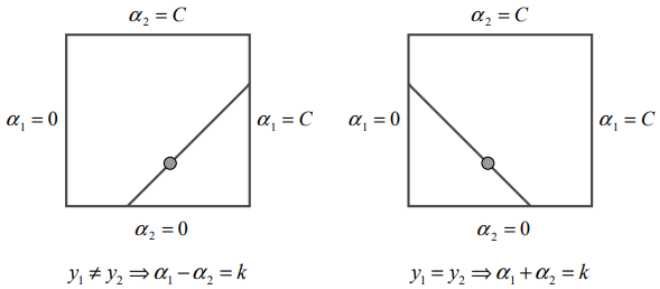

Figure 11: Visualisation of restriction placed on the two multipliers, here α is used instead of λ [12]

Notably, it is possible to express one multiplier in terms of the other. Thus, it is sufficient to manipulate only one multiplier, as the value of the second multipler follows from the first. This also creates a tighter restraint on the variables:

if $y_1 \neq y_2$:

$$\begin{cases} L = max(0, \lambda_2^{old} - \lambda_1^{old}) \\ U = min(C, C - \lambda_1^{old} + \lambda_2^{old}) \end{cases} \tag{19}$$

if $y_1 = y_2$:

$$\begin{cases} L = max(0, \lambda_1^{old} + \lambda_2^{old} - C) \\ U = min(C, C - \lambda_1^{old} + \lambda_2^{old}) \end{cases} \tag{20}$$

Now, in order to determine the optimal multiplier pair (λ_1, λ_2) the following steps are taken:

First, we compute the new optimal value for λ_2: [12]

$$\lambda_2^{new} = \lambda_2^{old} + \frac{y_2(E_2 - E_1)}{k} \tag{21}$$

with

$$k = x_1 \cdot x_1 + x_2 \cdot x_2 - 2 * x_1 \cdot x_2$$
$$E_i = (\sum_{j=1}^{n} a_j y_j x_j \cdot x_i + b) - y_i \quad ; \quad i = 1, 2$$

Note that E_i is not always computed as the model possesses an error cache. Afterwards, the obtained value for λ_2^{new} is then clipped to the bounds L and U determined in (19) or (20): [12]

$$\lambda_2^{new} = \begin{cases} L, & if \quad \lambda_2^{new} < L \\ U, & if \quad \lambda_2^{new} > U \\ \lambda_2^{new} else \end{cases} \tag{22}$$

Finally, we compute the value of λ_1^{new}: [12]

$$\lambda_1^{new} = \lambda_1^{old} + y_1 y_2 (\lambda_2^{old} - \lambda_2^{new}) \tag{23}$$

We repeat this for a random pair of multipliers until the conditions described in (17) are fulfilled.

Furthermore, for each iteration of the algorithm, the model has to compute the intercept b with the newly determined values:

$$p = y_1(\lambda_1^{new} - \lambda_1) \tag{24}$$
$$q = y_2(\lambda_2^{new} - \lambda_2) \tag{25}$$
$$b_1 = E_1 + p(\vec{x_1} \cdot \vec{x_1}) + q(\vec{x_1} \cdot \vec{x_2}) + b \tag{26}$$
$$b_2 = E_2 + p(\vec{x_1} \cdot \vec{x_2}) + q(\vec{x_2} \cdot \vec{x_2}) + b \tag{27}$$
$$b_3 = \frac{b_1 + b_2}{2} \tag{28}$$

with K being a kernel function. b_1 or b_2 are chosen when λ_1 or λ_2 respectively are non-bounded. If both are at the bound, we take the average of b_1 and b_2. The new intercept, along with the newly determined multipliers are then used to update the error cache, and the algorithm goes to its next iteration

To summarize, the SMO algorithm iterations consist of the following steps:

1) Choose the first multiplier
2) Choose the second multiplier that allows for maximal improvement
3) optimize the multipliers.
4) compute intercept

Kernels and the Kernel-Trick

Using the previously presented algorithm, it is now possible to train a support vector machine to classify linearly separable data containing outliers. Nonetheless, not being able to use SVMs for non-linearly separable data is quite a big restriction. Thankfully, through the use of something called a Kernel, it is also possible to train SVMs on non-linearly separable datasets.

Initially, when a dataset is not linearly separable, a function $\phi\ R^i \rightarrow R^j \quad with \quad i < j$ is used to map a data point into a higher dimension such that the data is linearly separable again. The SVM then uses the higher-dimensional points provided by ϕ to train and make predictions. Unfortunately, by mapping these data points into a higher dimension, the computational time for the dot product also increases.

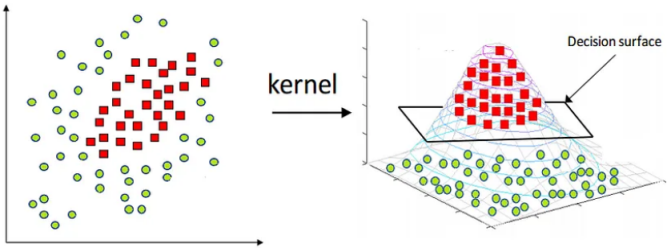

Figure 12: Mapping 2d points into 3 dimensions

This is where the Kernel Trick comes into play. The Kernel function manages to compute the dot-product of the higher dimensional points returned by ϕ using only the original points. As an example, define the following points x, z and the function ϕ.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad z = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix}$$

.

$$\phi(x) = \begin{pmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \end{pmatrix} \quad \phi(z) = \begin{pmatrix} z_1z_1 \\ z_1z_2 \\ z_1z_3 \\ z_2z_1 \\ z_2z_2 \\ z_2z_3 \\ z_3z_1 \\ z_3z_2 \\ z_3z_3 \end{pmatrix}$$

To be exact, ϕ returns a point whose coordinates are the different ways one can multiply two coordinate values. In fact, if a point x has size n, then $\phi(x)$ contains n^2 values. This causes a huge increase in the time needed to compute the dot product $\phi(x) \cdot \phi(z)$.

However, it is possible to express $\phi(x) \cdot \phi(z)$ in terms of

$x \cdot z$:

$$\phi(x) \cdot \phi(z) = \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j \tag{29}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j = \sum_{i=1}^{n}\sum_{j=1}^{n} (x_i z_i)(x_j z_j) \tag{30}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} (x_i z_i)(x_j z_j) = \sum_{i=1}^{n} x_i z_i \sum_{j=1}^{n} x_j z_j \tag{31}$$

$$\sum_{i=1}^{n} x_i z_i \sum_{j=1}^{n} x_j z_j = (\sum_{i=1}^{n} x_i z_i)^2 \tag{32}$$

$$(\sum_{i=1}^{n} x_i z_i)^2 = (x \cdot z)^2 \tag{33}$$

Let the Kernel K(x,z) thus be $(x \cdot z)^2$. The SVM can then be trained and make predictions having the advantage of higher dimensionally mapped data points without having the downside of slower computations by replacing the dot products in all the previous formulas with K(x,z).

It is possible to define one's own kernel functions; however, the most used Kernels are the Gaussian and polynomial kernels:

$$K_g(x, z) = \exp -\frac{|x - z|^2}{2\sigma^2} \tag{34}$$

$$K_p(x, z) = (x \cdot z + c)^d \tag{35}$$

4.3.1. The SVM model. This section presents the implementation of the SVM model. This section is only going to cover the methods used for data exchange, training, and prediction making.

First and foremost, before being able to make predictions or train a model, it is necessary to provide the model with data. Thus, the SVM model possesses a set_data method that takes two arrays. The first array X contains all the feature values of the data points, whereas the second array y contains the respective labels of the data points. The arrays are saved as attributes within the model object, and the arrays for the Lagrange multipliers, the weight vector, and the error cache are created.

Next, the $train$ method of the SVM model only works once data has been provided to the model using the previous method. Important to understand is that all multipliers are referenced using their indexes within the multiplier list stored within the model. The $train$ method is going to repeatedly choose a multiplier to optimize and pass its index to the $examinExample$ method, where the algorithm continues. The $train$ method chooses the multiplier in the following way: [12]

```
numChanged = 0
examineAll = True

while numChanged > 0 or examine All do
    if examine All then
        for all multipliers do
            numChanged += examineExample(mult_idx)
        end for
    else
        for all non−bounded do multipliers do
            numChanged += examineExample(mult_idx)
        end for
    end if

    if examineAll then
        examineAll = False
    else if numChanged == 0 then
        examineAll = True
    end if
```

The $train$ method prioritizes non-bounded multipliers as they are more likely to be non-optimized. This is because there usually are not many support vectors within a dataset. Furthermore, notice that $examineExample$ returns integers. More precisely, the method returns 1 when the multiplier has been updated and 0 otherwise.

The $examineExample$ method takes the index of a multiplier and searches for an optimal multiplier to perform coordinate descent on. The method first fetches the relevant information tied to the multiplier of the index. This includes the value and label of the multiplier, the features, and the prediction error. How that information is stored and passed onto the $coordinate\_descent$ method is up to the implementer. Nonetheless, the SVM model of the technical deliverable uses class attributes to save and fetch the multiplier's information. If the prediction error is lower than a certain tolerance and the multiplier is non-bounded, then the method returns 0 because the multiplier does not have to be updated. If this is not the case, $examineExamples$ tries to find a second multiplier that maximizes the absolute error $|E_1 - E_2|$ in order to determine a multiplier pair that is likely to minimize the objective function after modification. If the chosen multiplier does not cause the multipliers to update, new possible second multipliers are chosen randomly from all non-bound multipliers until a multiplier that updates both values is found. When this fails, the method tries one last time by choosing randomly from all available multipliers. Consequently, if $examineExample$ does not manage to update the multipliers, it returns 0 instead of 1.

In order to optimize two multipliers, $examineExample$ passes the chosen multipliers' indexes to the $coordinateDescent$ method.

First, the $coordinateDescent$ function checks that the two multipliers are distinct and returns if the multiplier's indices are identical. Afterwards, the function determines the lower

and upper bound that the multipliers can take using (19) and (20). If the bounds coincide, the function returns, as the multipliers would not change. Then, if the function has not already returned, the new value for the multipliers is determined using (21), (22) and (23). Finally, the new intersect b is calculated, and the error cache is updated. After completing these steps, the function returns True, informing the examineExample function that a multiplier pair has been successfully improved, which in turn informs the train function about the successful update.

This entire process of choosing a multiplier pair and trying to optimize it repeats until the train function passes through the entire list of multipliers without making any modifications.

4.3.2. The Model Manager. The model managing unit is responsible for organizing and managing the SVM models. When a managing unit is instantiated, it is expected to provide to the unit a name for the file in which a newly trained model will be stored or the name for the already existing record file of a model. Then, the managing unit can do its job once it initializes its environment with the provided data. First, a dataset has to be passed to the $provide\_data$ method. The $provide\_data$ method takes two arrays as input, one array for the features and one for the labels, and stores them as class attributes. Furthermore, the method traverses the array containing the label and determines the number of different labels. This allows the managing units $initialize\_environment$ method to know how many SVMs it has to instantiate.

Once everything has been set up, it is possible to request the managing unit to train a new model based on the provided data or to classify the provided data using an already trained model.

Regarding the training of the models, the $train\_models$ method iteratively takes an SVM model, passes the dataset along with a modified list of the labels list, and instructs the model to start its training process. To elaborate, because each SVM model is tasked to make {y, not-y} type of classifications, for each SVM, the labels list is modified such that all labels are either {+1,-1} where +1 stands for the class the model is tasked to recognize and -1 for all other classes. After training, the weights are stored in a text file.

```
--train_models--
weight_dict = {}

for each label do
    select label appropriate SVM model
    process the labels list to have a {−1,1} structure
    provide training data to SVM
    train SVM
    store the weights and multipliers in weight_dict
end do
store data within weight_dict in a file
```

If a model has previously been trained, it is also possible to request predictions from the model managing unit. In this scenario, the string passed to the managing unit represents the name of the file under which the trained model has been stored. The managing unit opens the record file and loads the necessary data into memory. This includes the weights, the Lagrange multipliers, and the training data. Once the models have been set up, the managing unit passes the data from the user-provided dataset to the models. The managing unit classifies the data points by determining which model returns a positive result on classification. These results are stored in a dictionary and returned at the end of the prediction process. This data is later saved in a file by the argument processing unit.

```
--make_predictions--
load SVM models
results = {}
outPut = {}
for each datapoint do
    for each SVM_model do
        outPut[SVM_model] =
            SVM_model.predict(datapoint)
    result[datapoint] = maxValue(outPut)
    outPut = {}
    end do
end do
return results
```

4.3.3. The Argument Processor. The argument processing unit has 2 important tasks: Determining the user's intentions and providing the model managing the unit with the appropriate instructions. The processing unit uses the 'getopt' library to parse the command line arguments given by the user. Specifically, the user can choose among 3 options: training an SVM model '-t', checking the accuracy of a trained model '-a' and making predictions using a trained model '-p'. Furthermore, the '-o' argument allows the user to specify the name of the file record of the SVM model to be stored or loaded. The analysis of the command line arguments is done by the $verify\_inputs$ method, which takes the command line arguments as input, determines what action the user wants to take, stores that information within itself, and returns the reference for the file record as well as the path to the dataset given by the user.

Moreover, to execute the $delegate\_task$ method, it is necessary to pass a model managing unit as input and inform the method what tasks it should instruct the model managing unit to execute.

```
--delegate_task(model_manager, data_path)--
X,y = process_data(data_ref)
model_manager.provide_data(X,y)
model_manager.initialize_environment()

if self.train then
    self.initiate_training(model_manager)
if self.acc then
    self.initiate_acc_check(model_manager)
if self.predict then
    self.initiate_prediction(model_manager)
```

4.3.4. Program evaluation. The following section presents the performance of the SVM model produced for this project. The model is going to be tested on training speed as well as its prediction accuracy. Moreover, the SVC model provided by the sk-learn library [11] is used to benchmark the SVM performance. 3 datasets are going to be used to test the produced model: the Iris dataset [11] provided by the sklearn library, a dataset constituting of a large circle of dots and a smaller circle within the larger circle and, finally, a dataset of 32x32 pictures of letters. Important to note is that the testing of the programs is done on the iris-supercomputer, provided by the University of Luxembourg, which uses a Xeon E5-2680v4 processor [14].

Iris Dataset

In short, the training of a model took, on average, 9 seconds, resulting in an accuracy of 97%. To compare, on average, sk-learns SVC model [11], manages to finish its training in 0.7 seconds with an 100% accuracy rate. This is quite a large gap in performance as this project's model is 10 times slower.

Two cricles dataset

Next, the SVM model is tested using the 'make_circles' function provided by the sk-learn library [11]. The goal is to observe how the training time varies depending on the number of training data set sizes.

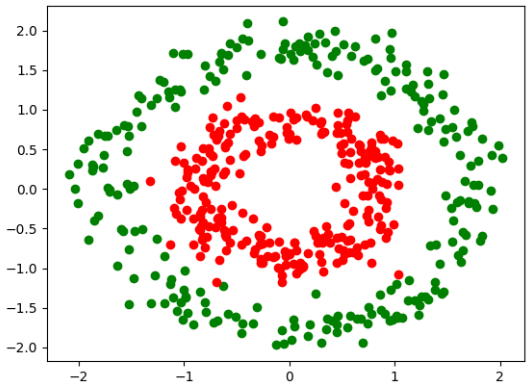

Figure 13: Example of a data set created with 'make_circles'

The model, as well as the sk-learns SVC, were tested using datasets of sizes ranging between 100 and 2000. Because the training algorithm for the SVM used has a heuristic component, the training processes have been repeated multiple times, and an average execution time has been taken. Notice that the scaling of training time with regard to the data set size is poor compared to the SVC model. Furthermore, even though the execution time was measured multiple times on different datasets, there still exist spikes in the graph representing the models performance. 14 Otherwise, the accuracy of the trained SVM model always ranged between 95% and 97%.

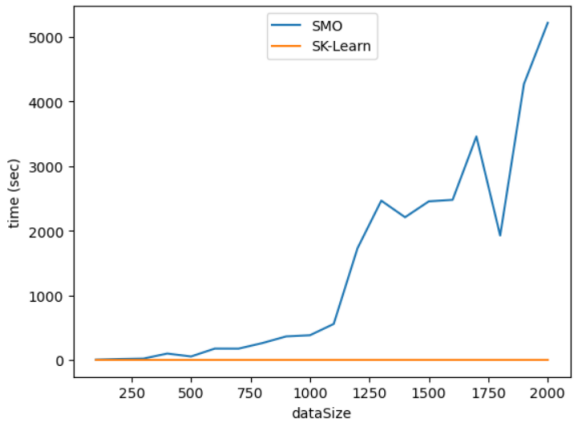

Figure 14: SVM model vs SVC model

Alphabet dataset

Finally, the model is tested using a data set containing 32x32 images of letters in different fonts [7]. First, the training time for the SVM model becomes quite large as the number of features are 1024. The execution time goes from a few seconds with 50 Images for the letters A and B each to an hour when provided with 350 images each.

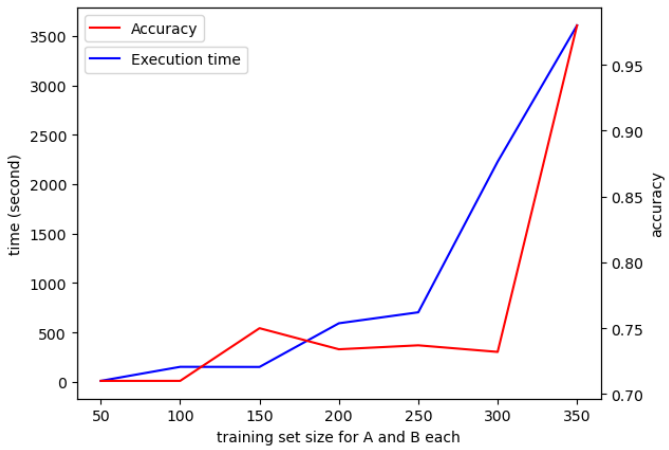

Figure 15: Execution time and accuracy classifying pictures of the letter A and B

Regarding the accuracy of the model, the model remains

consistent with its accuracy throughout testing, giving a 75% precision average when classifying pictures of the letters A and B.

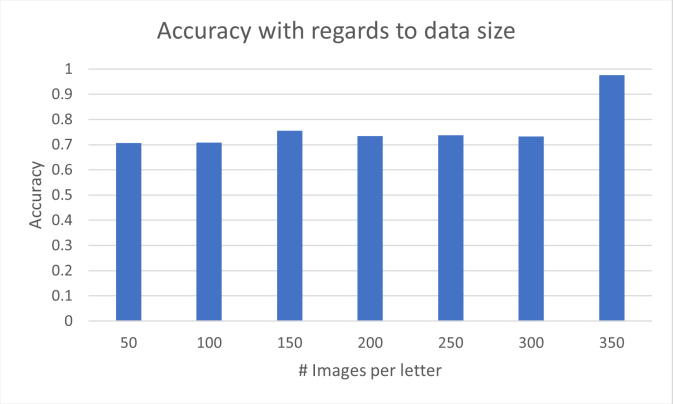

Figure 16: Accuracy classifying pictures of the letter A and B

Testing has also been done using more letters. However, because the model is quite slow, the testing has been limited to 250 sample images of the letters from A to F. It is safe to say that there is a lot of room for improvement. The accuracy provided by the model is quite poor, as the accuracy just gets over 55 percent.

Possible Issues
First and foremost, it is difficult to figure out whether the accuracy issue is the model itself or the lack of training data. Generally speaking, the success of the model with smaller datasets pushes the idea that the issue is the lack of training data provided to the model. Unfortunately, the increase in training data increases the time of training manifold. Furthermore, seeing that SMO has a heuristic part to its algorithm, it is probably that the cache-hit rate is quite poor as after each optimization of two Lagrange multipliers, two different ones are taken that are not necessarily in the same cache line as the previous ones. Thus, analysing how data is accessed during execution could also provide a significant increase in speed. Finally, in hindsight, another probable issue could be the lack of data preprocessing done for the testing of the models, which can have a negative impact on the training time.

4.4. Assessment

The goal for this technical deliverable was to create a program that is capable of receiving data and training an SVM model or making predictions using an already trained model. In fact, a working SVM model, as well as additional infrastructure to allow for multi-classification using SVMs, has been created. The program is capable of taking in data and training a model that is capable of making correct predictions with a certain degree of competency. However, the program is far from perfect because the program falls short when compared to other already existing models. To elaborate, the training time grows very fast with the size of the dataset, and the accuracy of the model suffers because of it. Possible causes for this could be poor usage of Python libraries. Furthermore, the SMO algorithm is still used today in the LIBSVM library [2]. Nonetheless, the LIBSVM model is written in C++, which can probably take better advantage of the 'Sequential' part of the algorithm because, in a relatively slower language like Python, it would be advantageous to parallelise the training process. To conclude, this project successfully covers the development of the SVM model, its managing unit and the argument processing unit, going in-depth about how an SVM model works as well as how the model is trained. Nonetheless, the resulting program's performance is not on par with already developed models and requires further research.

Acknowledgment

This endeavour would not have been possible without Dr Ezhilmathi Krishnasamy, who provided invaluable guidance and feedback during this project.

Furthermore, thanks also go to the BiCS management and education team for the amazing work done.

Lastly, the experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [**?**] – see https://hpc.uni.lu

5. Conclusion

This bachelor semester project concludes with this section. All in all, the objectives of both the scientific and technical deliverables have been accomplished. Nonetheless, there are still many aspects of message-passing interfaces, as well as machine learning, that have not been covered yet. Therefore, the reader is encouraged to continue their own research regarding these topics. To conclude, the project's scientific deliverables have introduced me to the topic of distributed computing as well as the use of MPI, and the technical project has taught me a lot about data processing, the SVM model for classification, and multivariable calculus.

6. Plagiarism statement

This 350-word section without this first paragraph must be included in the submitted report and placed after the conclusion. This section does not count in the total word quantity.

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg, I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.

- My report will be checked for plagiarism, and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issues.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
    1) Not putting quotation marks around a quote from another person's work
    2) Pretending to paraphrase while, in fact, quoting
    3) Citing incorrectly or incompletely
    4) Failing to cite the source of a quoted or paraphrased work
    5) Copying/reproducing sections of another person's work without acknowledging the source
    6) Paraphrasing another person's work without acknowledging the source
    7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing)
    8) Using another person's unpublished work without attribution and permission ('stealing')
    9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

    Auto- or self-plagiarism, that is, the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[2] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[3] Tristan Fletcher. Support vector machines explained. *Tutorial paper*, pages 1–19, 2009.

[4] Florian Janetzko. Message-passing interface-selected topics and best practices. Technical report, Jülich Supercomputing Center, 2014.

[5] George Karniadakis and Robert M Kirby. *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation*, volume 2. Cambridge university press, 2003.

[6] Yuh-Jye Lee. Sequential minimal optimization (smo), 2017.

[7] Thomas Lin. Alphabet characters fonts dataset, 2020.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[9] Frank Nielsen. *Introduction to HPC with MPI for Data Science*. Springer, 2016.

[10] Lecture 32: Introduction to mpi i/o. Lecture.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[13] Ming-Chit Tam, Jonathan M Smith, and David J Farber. A taxonomy-based comparison of several distributed shared memory systems. *ACM SIGOPS Operating Systems Review*, 24(3):40–67, 1990.

[14] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.

7. Appendix

7.1. Proof

7.1.1. Dual of SVM optimization problem. First, let us develop (9), multiplying the terms with each other:

$$\frac{||w||^2}{2} + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\lambda_i y_i w \cdot x_i + b\sum_{i=1}^{N}\lambda_i y_i + \sum_{i=1}^{N}\lambda_i + \sum_{i=1}^{N}\lambda_i\xi_i - \sum_{i=1}^{N}v_i\xi_i$$

Substituting (14), getting rid of a term:

$$\frac{||w||^2}{2} + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\lambda_i y_i w \cdot x_i + \sum_{i=1}^{N}\lambda_i + \sum_{i=1}^{N}\lambda_i\xi_i - \sum_{i=1}^{N}v_i\xi_i$$

Using (15), to remove further sums:

$$\frac{||w||^2}{2} - \sum_{i=1}^{N}\lambda_i y_i w \cdot x_i + \sum_{i=1}^{N}\lambda_i$$

Finally, substituting (13):

$$\frac{1}{2}(\sum_{i=1}^{N}\lambda_i y_i x_i)^2 - \sum_{i=1}^{N}\lambda_i y_i \sum_{j=1}^{N}(\lambda_j y_j x_j)\cdot x_i + \sum_{i=1}^{N}\lambda_i$$

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i x_j - \sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i x_j + \sum_{i=1}^{N}\lambda_i$$

Resulting in:

$$\sum_{i=1}^{N}\lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j y_i y_j x_i x_j$$

7.2. Diagramms

[h]

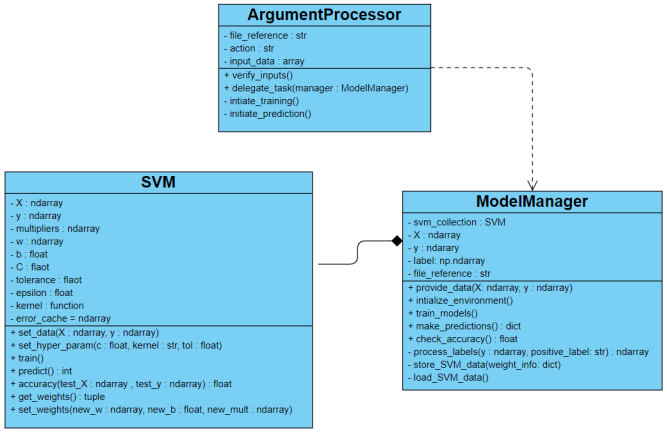

Figure 17: UML diagram for the SVM program