Using Iterative MapReduce for Parallel Virtual Screening

Laeeq Ahmed, Ake Edlund, Erwin Laure
Department of HPCViz,
Royal Institute of Technology, Stockholm
Email: laeq@kth.se, edlund@pdc.kth.se, erwinl@pdc.kth.se

Ola Spjuth
Department of Pharmaceutical Biosciences,
University of Uppsala, Uppsala
Email: ola.spjuth@farmbio.uu.se

Abstract—Virtual Screening is a technique in chemoinformatics used for Drug discovery by searching large libraries of molecule structures. Virtual Screening often uses SVM, a supervised machine learning technique used for regression and classification analysis. Virtual screening using SVM not only involves huge datasets, but it is also compute expensive with a complexity that can grow at least up to $O(n^2)$. SVM based applications most commonly use MPI, which becomes complex and impractical with large datasets. As an alternative to MPI, MapReduce, and its different implementations, have been successfully used on commodity clusters for analysis of data for problems with very large datasets. Due to the large libraries of molecule structures in virtual screening, it becomes a good candidate for MapReduce. In this paper we present a MapReduce implementation of SVM based virtual screening, using Spark, an iterative MapReduce programming model. We show that our implementation has a good scaling behaviour and opens up the possibility of using huge public cloud infrastructures efficiently for virtual screening.

Index Terms—MapReduce, Big Data, Spark, Chemoinformatics, Parallel SVM

I. INTRODUCTION

As a result of the recent advancements in information technology and instrumentation, scientists produce massive data sets at an extensive and increasing rate. Examples of scientific problems generating large datasets are next-generation DNA Sequencing [1] in Biomedical computing, next generation synoptic sky surveys [2] in Astronomy and weather research and forecast [3] in Meteorology. While today’s scientist are privileged to have these huge data sets to study, the increasing size, complexity and structure of this data, creates novel challenges for the scientists to process and analyze the data. With the data being complex and of unstructured nature, conventional relational databases are not suitable for manipulating this data. Similarly conventional data mining techniques used previously by the scientists now struggle to cope with the large data sets. With all these problems at hand, the need and importance of parallel applications becomes more apparent and important. A number of parallel programming frameworks have been introduced in the recent past e.g. threads, MPI, OpenMP and MapReduce. Of these MPI has been the most preferred parallel framework for a long period of time. With its diverse set of constructs, MPI allows to communicate in different ways e.g. point-to-point communication or collective communication.

In 2005, Google produces its famous paper [4] on the MapReduce framework, explaining the idea of using two basic functions, map and reduce, for the processing of large datasets. Comparing MapReduce with MPI, MPI is more flexible and can be used in a variety of applications. However, when the size of the data is large, MPI becomes complex to use and is less practical [5]. MapReduce especially shines when the size of the datasets is large and computation to perform is relatively simple. In MapReduce all the Map processes and Reduce processes execute independently and have weak synchronization. Thus if a process fails, it can be easily rerun without affecting any other process. MapReduce is mostly used for data-intensive tasks but a number of studies [6], [7] show that it can be extended to compute-intensive problems, too. The significant advantage of MapReduce is that it provides parallelization, fault-tolerance, data distribution and load balancing in a single framework, all transparent and already done for the user. Furthermore, by using MapReduce we open up the possibility to use huge public cloud infrastructures for these kinds of problems. Hadoop [8] is the most widely used open source implementation of MapReduce. It has been extensively used at organizations like Facebook, Amazon etc. Another well-known alternative is Dryad [9], produced by Microsoft.

In eScience there is an increasing number of applications, including data mining, web ranking, graph processing, model fitting etc. that are based on iterative algorithms. The standard MapReduce framework is based on a acyclic data flow model and does not have built-in facilities for iterative programs. Manually programming and chaining multiple MapReduce tasks together, by providing the output of the previous MapReduce as the input of the next MapReduce can accomplish iterative programs. This use of MapReduce to achieve iterative analysis of large data sets results in a number of limitations [11]. Before each MapReduce iteration, the data has to be reread and reprocessed, resulting in additional cost in network bandwidth, I/O and CPU. This is the case even when most of the data per iteration remains unchanged. Secondly, within each MapReduce iteration, we need to check for convergence to decide if we should end the iterative process. We also might have to add extra MapReduce steps in each iteration to validate the termination condition, causing cost of additional tasks, reload of additional data and data movement over the network. Due to these issues a number of alternative
frameworks have been devised for iterative applications e.g. Spark [10], Twister [5], and Haloop [11].

Spark [10] is an open source implementation of MapReduce, used for large data analytics. In addition to Spark's iterative implementation of MapReduce, it has a number of novel features including in-memory computing and fault tolerance, using Resilient Datasets (RDDs). With in-memory computing, we can store datasets in memory resulting in considerable higher access rates. If the size of the analyzed datasets is bigger than memory, the additional data will be automatically stored on disk. RDDs bring in innovative fault tolerance into Spark by making use of the concept of lineage. Through lineage RDDs remember all the transformations from which they have been derived and on failure, can roll back to a state from which they can be reproduced.

SVMs were developed by [12] for two-group classification problems and for some years it has become a popular tool for solving problems in classification, regression and novelty detection. For many applications SVMs give better results than other techniques, however they are more resource demanding and compute intensive. For example, due to the computational complexity of SVM, which is at least $O(n^2)$ [13], SVM based virtual screening can be very time consuming. Studies have been performed to increase the efficiency of SVM, for example through novel parallel implementations. [14] outlines the idea of parallelizing SVM by dividing data into small portions and then use SVM to solve each small portion in parallel. Most of these parallel implementations are based on MPI rather than MapReduce.

On the basis of the above discussion, we believe that iterative MapReduce is a good choice for SVM based virtual screening. In this study, we implemented parallel SVM based virtual screening model using Spark that runs on commodity clusters. In our implementation the data and relevant libraries are distributed on the nodes of a cluster, performing virtual screening, separately and parallel on each dataset. While LibSVM [15] is used for training and classification of molecule structures, the implementation is performed using CDK [16] and Java.

Rest of the paper is organized as follows: In section 2, we present some details about SVM and its usage. In section 3, the Spark programming model is described. In section 4, we provide details about our implementation of virtual screening. Section 5 describes our experiments and results, followed by conclusions and future work in section 6.

II. SUPPORT VECTOR MACHINES AND APPLICATIONS

Support vector machines (SVM) [12] have been widely used for classification and regression problems. A SVM includes a training algorithm that uses training examples, which are already marked to one class or another, to build an SVM model that predicts the new examples into one category or another. A SVM model represent training examples as points in space, dividing these examples into categories by a clear gap which is as wide as possible. In figure 1, hyperplane $H_1$ separates the input data into two categories by maximum-margin. Though $H_2$ also separates the input data, but only with less or no margin. Once training examples are categorized, new examples are then predicted on the basis that on which side of the gap they fall in the same space.

Other than linear classification, SVM can also perform non-linear classification using the kernel trick originally proposed by [17]. Most of the Non-linear classification algorithm is same as linear SVM. The only difference is that a nonlinear kernel function is used instead of every dot product, thus enabling the algorithm to fit the maximum-margin hyperplane in a transformed feature space. Figure 2 illustrates non-linear classification. SVM has been used in different areas. In bioinformatics, they have been used to identify small organic molecules [18] and to distinguish epoxide hydrolases (EHs) from other enzymes and to classify its subfamilies using its primary protein sequences [19]. [20] provides its usage for pattern recognition. SVM has also been used in chemoinformatics. In [21] and [22] are two examples where SVMs have been successfully used in the field of chemoinformatics.
A. Virtual Screening and LibSVM

Virtual screening (VS) is aimed at filtering large collections of chemical structures to identify candidates that are likely to bind to a drug target [23]. Here we use the signature descriptor [24] to describe the chemical structures numerically and SVM models to rank candidate structures according to the desired predicted chemical properties, including inhibition of cancer cell lines as well as partition coefficient (clogP) and solubility (logS), and also safety models such as HeRG and AhR inhibition as described in [25] and [26].

LibSVM is the most used and efficient implementation of SVM. The tool provides facilities of support vector classification (SVC), support vector regression (SVR) and Distribution Estimation (One-Class SVM). In this paper we have used LibSVM for prebuilt classification and regression models (see above) applied on the over 21 million molecules in the Zinc database [27] available in SDF(structure-data) format.

III. SPARK: ITERATIVE MAPREDUCE

Spark [10] is an iterative low latency cluster computing system used for Big Data Analytics. It is compatible with apache Hadoop and allows us to read data from any Hadoop supported storage system or file format e.g. HDFS, Amazon S3, Sequence Files etc. With iterative and in-memory dataset processing, it is on average 10 times faster [10] than Hadoop. Spark can be specifically beneficial for virtual screening for the reasons that normally virtual screening is done on large datasets and Spark is good at handling large datasets. Also in some cases, scientists want to test the same datasets with many models. In such cases, Spark iterative MapReduce can be especially useful keeping the datasets in memory to process it iteratively using various models. Keeping datasets in memory avoiding repeated re-reading from disk can save plenty of processing time. More details on Spark usage and constructs are as follows:

A. Spark Programming Model

To use Spark, developers have to write driver programs. In the driver program, we can give the overall flow of control for our application and apply different operations in parallel. The facilities provided for parallel programming are as follows:

1) Resilient Distributed Datasets: A resilient distributed dataset (RDD) is an immutable collection of objects partitioned across a set of machines. If one of the partitions is lost, it can be restore by backward transformation, through lineage, thus providing fault tolerance without replication. RDDs are created by reading files from a shared file system e.g. Hadoop Distributed File System(HDFS). These RDDs can later be transformed by applying various available operations e.g. flatmap. Optionally, RDDs can be kept in memory for iterative in-memory processing using cache operation.

2) Parallel Operations: Spark allows us to perform different parallel operations on the RDDs. E.g. reduce can be used to reduce the elements of the RDD from parallel machines or collect which returns all elements of the datasets in the RDD to the driver program.

3) Shared Variables: Other than normal variables, spark provides special type of shared variables for two special cases. When there is huge size data that is only readable, broadcast variables can be used. Broadcast variables are only distributed once to the workers, rather than with every single parallel operation. The second situation is when worker nodes need some variable, which the master node can only read. E.g. if we want to implement counters in MapReduce, these counters can be used for parallel sums with a more imperative syntax. Such variables are named as accumulators. Accumulators are easier to make fault tolerant, as they are add-only.

IV. VIRTUAL SCREENING ON SPARK IMPLEMENTATION

For virtual screening, we have SDF files, which contain the molecule structure. We classify these molecules using LibSVM that creates models and then predict them using these models. This is basic investigation of Spark for virtual screening where we only use two SVM models to predict molecules structures.

A. Tools and Infrastructure

In this study, we have used an OpenNebula [28] based private cloud infrastructure. OpenNebula is cloud creation and management software that allows you to launch, delete, and clone VMs and images. The user can either use OpenNebula command line interface (CLI) or web interface with GUI to control the cluster environment. During our experiments we found that currently the GUI based interface is not fully stable and we used CLI for our cluster management.

On top of our Open Nebula private cluster, we used the Hadoop Distributed File System(HDFS) [29] for data storage as shown in figure 3. HDFS is part of the Apache Hadoop framework [8] and different than other distributed file systems in terms that it is fault tolerant and can be used on commodity hardware. It allows quick access to application data and is most suitable when applications have very large datasets. In addition the datasets can be replicated for fault tolerance requirement. For data processing and analysis, we used Spark, as already discussed in section III. Based on these services, we
developed our virtual screening application further described in the following section.

B. Architecture of Parallel Virtual Screening

Our parallel virtual screening is based on master-worker architecture as shown in figure 4. The master machine acts as NameNode of HDFS and the worker node is used as DataNode of HDFS. Once the SDF files get copied to our private cluster, NameNode divides each SDF file into blocks; the data blocks are distributed and kept at DataNode whereas NameNode only contains the metadata about the dataset stored at each DataNode. NameNode also have file system namespace and controls how worker nodes can access the files. The number of blocks depends upon the size of the file. The number of blocks are calculated by the size of the file divided by the HDFS block size, which is normally 64MB in older versions 128MB in newer versions. The data blocks contain the records, which are the smallest readable unit by the workers. In our case one molecule structure acts as a record.

Once the dataset gets distributed on the cluster, we process these datasets using Spark MapReduce with Spark master daemons and worker daemons running at Master and Worker node respectively as shown in figure 4. To transfer our application code and all the dependencies including LibSVM and CDK, we create a single jar file using Apache Maven [30]. The jar file is then provided to Spark through SparkContext (explained later in section IV-C), which deploys it to all the worker nodes. The benefit is that all the dependencies are local to the worker nodes and don’t have to remotely fetch from network, thus reducing the overall processing time. The worker nodes process these datasets using MapReduce, Each worker node has mappers and reducers. Since each block is assigned to one mapper the parallelism and number of mappers depends on the number of input blocks, whereas the number of reducers can be changed through the driver program. Once the worker nodes complete the processing, the master node gets back the final result.

C. Work Flow of driver program

The parallel virtual screening pseudo code based on spark is as follows:

Before executing the driver program, we make some preparations. Master and workers nodes are specified to Spark and HDFS in their specific configuration files by providing corresponding IP addresses. There are many other configuration variables of Spark and HDFS that need to be set in their respective configuration files. For Spark we can e.g. set a Spark worker memory that will be available during processing by setting the Spark_WORKER_MEMORY variable. Using a distributed file system, as in our case HDFS, the data is then distributed over the worker nodes before processing.

In addition we created a RunPrediction class which acts as the driver program. In this class, we control the overall parallel processing of the datasets. In the initialization process, we read and initialize predictive models into memory using SignLibsvmModel class, which we have created separately. During initialization, we also set different spark system properties. By default Spark uses all the cores available on the machine, while if the user want Spark to only use 4 cores, this is achieved by:

```
System.setProperty("spark.cores.max", "4");
```

Spark uses SparkContext to access a cluster. In SparkContext, we provide the master node IP and the jar file path that we need to pass on to worker nodes. Next we read SDF files into RDD using SparkContext. Spark allows us for reading text files or those supported by Hadoop. SDF files cannot be read by built-in .textFile() method which supports records based on single line. For reading multline molecules structure, we need to have custom input format by extending the Hadoop FileInputFormat class and implement a record reader. In our case, we used the SDFInputFormat [31] class to read SDF files. In the SDF file, each molecule is separated by $$$$ and newline. The following code shows how we read the SDF files, where the endTag represent the end of a molecule.
byte[] endTag = "$\$$\$$\n".getBytes( );
boolean status = readUntilMatch(endTag,true);

Once we have created RDD, which contains SDF files, we apply the map operation on all the RDDs. By applying the map operation a function is applied to all elements in the RDDs. Internally this function is executed separately in parallel on all the mappers on the worker nodes as shown in figure 6.

![Fig. 6. Molecule Prediction with MapReduce](image)

In the map operation, we used CDK to read molecules into IAtomContainer object by parsing the string representation of the SDF record. Once parsed, we can manipulate these molecules normally as done using CDK without parallelism and MapReduce. We calculate the signature of the molecules and then compare these molecules to the SVM models to find the predicted molecules. The mappers emit a key, value pair of molecules and associated value of 1 or 0, where 1 represent a predicted molecule and 0 otherwise.

```java
if (hergRes!=1) return new Tuple2(mol,1);
else return new Tuple2(mol,0);
```

We then apply filter operation on RDD to filter out the molecules with value 1. After filtering, reduce operation is applied to get the aggregate successful molecules by reducing all the 1’s.

```java
Integer total = winners.reduce(
    new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer a, Integer b) {
        return a + b;
    }
});
```

V. EXPERIMENTS AND RESULTS

To evaluate the performance of our implementation, we performed some experiments. In the parallel virtual screening experiment environment, we launched and used virtual machines on our OpenNebula based cluster. All these virtual machines are based on AMD 64bit architecture. Each VM acts as either a master node or a worker node. Each of the worker nodes have an internal memory of 2GB and with 2 cores each whereas the master node has an internal memory of 10GB and 2 cores for processing. The master node normally doesn’t need much memory but when the number of worker nodes increases, it needs to have more memory to keep record of all the worker nodes. Each node is installed with Debian OS.

A. Parallel Virtual Screening Scaling

In this experiment we evaluate the performance of virtual screening with increasing number of nodes/parallelism. The dataset used for this experiment is 10GB (approximately) of SDF files. All the files are equal in size, each of 564MB. The details about the experiment are illustrated by figure 7.

![Fig. 7. Parallel Virtual Screening Scaling](image)

We started with 10 worker nodes, increasing 4 nodes on each test. Initially, with the increasing number of nodes, the time is decreasing almost linearly, where with 10 nodes virtual screening is completed in 49 minutes and with 26 nodes, its completed within 25 minutes. This suggests that virtual screening can perform better with increasing number of nodes. After 26 nodes, the time does not decrease as rapidly and once we increased nodes to 38, it flattens out to 21 minutes. This shows that parallelization can be done to certain limits.

B. Performance with changing input split size

In this experiment, we evaluate how the parallel virtual screening performs when we change the input split size. The number of nodes we used for this experiment are 22 i.e. 44 cores. During the evaluation we increased the total size of the data, with a total number of 5 different sizes. We tested our implementation with two different input split sizes, 64MB and 128MB, as shown in figure 8.

![Fig. 8. Changing Input Split](image)

In this test, we found that increased input split size slows down the execution time of our implementation. The reason is
that increased input split size decreases the parallelism since
the number of mappers is inversely proportional to the input
split size. With 128MB, we have fewer mappers than with
64MB split size and thus less parallelism. This experiment also
answers why scaling was limited to a certain limit in the earlier
experiment where our input split size was 64MB, and we could have further parallelize it by decreasing the input
split size to 32MB. This also shows that if the size of data is
big, one need to have bigger input split size and vise versa.

VI. RELATED WORK

We see some studies where MapReduce based SVM has
been used for different purposes. While most of these studies make use of standard MapReduce, [32] is a study, which
uses another iterative MapReduce implementation, Twister, to evaluate possible improvements of the performance of parallel
SVM. In this study, they showed that the computational time
could be greatly reduced by using the proposed twister based parallel SVM model.

Another Study [33] is about using Hadoop MapReduce
to implement Docking of Large-Scale Virtual Screening. In this study, authors explain how they stored small molecules in
HDFS, and then applied MapReduce framework to it. In the
Map Function, they executed molecular docking in parallel. In
the Reduce function, they collect the results. They also suggest
that the number of data blocks should be more than the number
of DataNodes, so that every DataNode have blocks to process.

VII. CONCLUSIONS AND FURTHER WORK

In this study, we used Spark, an iterative MapReduce
programming model, for implementing SVM based virtual
screening. We showed how HDFS and Spark can be used to
combine in distribution to process data in parallel, providing a satisfactory scaling behaviour and efficiency. While traditionally SVM based applications are implemented in MPI,
we showed that MapReduce is a viable alternative that opens
the path for efficient exploitation of large-scale public cloud
infrastructures for virtual screening.

This was a basic study where we used only two SVM
models to evaluate the ability of Spark for virtual screening.
Based on this study, we will use Spark in future studies to
implement multi-model virtual screening.

ACKNOWLEDGMENT

This work is partly funded through the Swedish e-Science initiatives eSSENCE and SeRC.

REFERENCES

[1] A. McKenna et al., _The Genome Analysis Toolkit: A MapReduce frame-
work for analyzing next-generation DNA sequencing data._, Advance July
[2] M. J. Graham et al., _Data challenges of time domain astronomy._, Dis-
tributed and Parallel Databases October 2012; Volume 30, Issue 5-6, pp
371-384.
[3] Y. Wu et al., _Parallel Processing on WRF Meteorological Data Using
MPICH._ Sixth International Conference on Internet Computing for
Science and Engineering (ICICSE), 2012.
Large Clusters._, OSDI’04; Sixth Symposium on Operating System
’10 Proceedings of the 19th ACM International Symposium on High
Performance Distributed Computing Pages 810-818
Performance Parallel Computing with Clouds and Cloud Technologies,
Cloud Computing and Software Services: Theory and Techniques, ed:
CRC Press (Taylor and Francis).
[9] M. Isard et al., _Dryad: Distributed Data-Parallel Programs from Se-
quential Building Blocks._, EuroSys07, March 2007, Lisbon, Portugal.
Copyright 2007.
[10] M. Zaharia et al., _Spark: Cluster Computing with Working Sets._, Hot-
[11] Y. Bu et al., _HaLoop: efficient iterative data processing on large clusters,_
Proceedings of the VLDB Endowment, Vol. 3, No. 1 Copyright 2010
VLDB Endowment 2150-8097/10/09
[12] C. Cortes, V. Vapnik, _Support-Vector Networks._ Machine Learning, 20,
[13] A. Bordes et al., _Fast Kernel Classifiers with Online and Active
[14] H. P. Graf et al., _Parallel support vector machines: the Cascade SVM._,
ACM Transactions on Intelligent Systems and Technology (TIST) Volume
2 Issue 3, April 2011 Article No. 27.
[17] M. A. Aizerman, Mark A. et al., _Theoretical foundations of the potential
function method in pattern recognition learning._, Automation and Remote
Control 25: 821837.
[18] E. Byvatov, G. Schneider, _Support vector machine applications in
hydrolases recognition and classification_, J Zhejiang Univ Sci B. 2006
Pattern Recognition: A Survey_, SVM 2002, LNCS 2388, pp. 213-236,
2002.
[21] K. Heikamp et al., _Potency-directed similarity searching using P2P
[22] R. N. Jorissen, _Virtual screening of molecular databases using a support
[23] M. Lill, _Virtual screening in drug design._, Methods Mol Biol 993
(2013), 112.
valence sequences in qsar and qpr studies._, Journal of Chemical
Information and Computer Sciences 43 (2003), 407720.
[25] O. Spjuth et al., _Integrated decision support for assessing chemical
[26] O. Spjuth et al., _ibioclipse-r: integrating management and visualization
of life science data with statistical analysis._, Bioinformatics 29, 2 (Jan
2013), 2869.
[27] J. J. Erwin, _A free tool to discover chemistry for biology._, Journal of
Chemical Information and Modeling 52, 7 (2012), 17571768.
[31] R. Goha, _Hadoop and SD Files._ Available at http://blog.gushui.net/?p=293
[32] Z. Sun, G. Fox, _Study on Parallel SVM Based on MapReduce._ The
2012 International Conference on Parallel and Distributed Processing
Techniques and Applications
[33] J. Zhao et al., _Hadoop MapReduce Framework to Implement Molecular
Docking of Large-Scale Virtual Screening._, 2012 IEEE Asia-Pacific
Services Computing Conference.