Structure-Based Virtual Screening in Spark

Marco Capuccini

Degree project in bioinformatics, 2015
Examensarbete i bioinformatik 45 hp till masterexamen, 2015
Biology Education Centre and Department of Pharmaceutical Biosciences, Uppsala University
Supervisor: Ola Spjuth
Abstract

Structure-Based Virtual Screening (SBVS) is a method used in drug discovery in order to search leads for certain target receptors. In the past decades high-throughput methods were used in order to build huge molecular libraries. ZINC and eMolecules are two high-throughput virtual molecular libraries that count million of molecules. Therefore, SBVS is computationally heavy for such libraries. However, SBVS is a trivially parallelizable task. In the past Message Passing Interface (MPI) has been used in order to parallelize SBVS. The main disadvantage of MPI based SBVS is that organizations need High Performance Computing (HPC) facilities in order to run it. Furthermore, MPI does not offer much more than a message passing interface. Therefore, MPI-applications are difficult to implement and maintain. Big data analytics on inexpensive hardware was pioneered by Google’s MapReduce programming model and parallel implementation. MapReduce applications have the property to be out of the box fault tolerant and scalable. Therefore, these applications are suitable to run in the cloud, which is in general cheaper than HPC. Even if Google’s MapReduce is legacy, many open source implementations such as Apache Hadoop are available. However, MapReduce is not a general purpose programming model and does not easily apply to SBVS. The Spark framework is an open source project, and it represents an evolution of Google’s MapReduce. The Spark programming model is more flexible than MapReduce and it gracefully applies to SBVS. Furthermore, Spark-applications are still out of the box fault tolerant, scalable and cloud runnable. Using Spark we implemented a tool for massively parallel SBVS. Our tool allows the user to define cloud-ready pipelines for high-throughput SBVS with few lines of code. The tests we performed in our private cloud show that our tool is simple to use and it scales well. Therefore, here we show how Spark can be used for massively parallel pipelining, and more in general how Big Data analytics can be beneficial in life science.
Structure-Based Virtual Screening in Spark

Popular science summary

Marco Capuccini

We live in the Big Data era. While producing and storing data is becoming cheaper and faster, we still have the problem of extracting meaningful information from it in reasonable time. Internet and social networks mostly account for the data storage and processing demand. For instance, YouTube statistics press reports that 300 hours of new video are uploaded every minute in its infrastructure. The need of processing such huge datasets represents a new challenge in computer science, and it makes obsolete all of the data analytics methods that we used to adopt.

Only in 2008, Google was already able to process 20 thousand terabytes per day. In order to do so, Google introduced the MapReduce (MR) parallel computing model. Instead of relying on expensive hardware, MR manages hardware failures and minimizes the network communication at software level. Therefore, the MR application can run on inexpensive computer clusters, reducing the cost of data analytics.

Spark is an open source project that represents an evolution of Google’s MR. It provides a more flexible programming model, allowing to solve a broader range of problems in distributed fashion. Furthermore, Spark applications still manage hardware faults and minimize network communication on software level, allowing to use cheap hardware.

Structure-based virtual screening (SBVS) is an in silico method that is aimed to search leads for a target receptor in a virtual molecular library. High-throughput methods in structural biology allowed to produce massive molecular libraries in the past decade. Therefore, since those libraries contain tens of millions of molecules, SBVS can nowadays be seen as a Big Data analytics problem.

In this study we developed a tool for SBVS in Spark. Furthermore, performing some experiments in our private cloud, we showed that Spark-based SBVS scales well. This open-ups SBVS to those organizations that do not own high performance computer facilities. For instance, an organization could start by using a small library and few cloud resources, and then scale to lot of cloud resources and high-throughput libraries as the business grows.

Degree project in Bioinformatics, Master of Science (2 years), 2015
Biology Education Center and Department of Pharmaceutical Biosciences
Supervisor: Ola Spjuth
Contents

Abbreviations 6

1 Introduction 7

2 Aims 8

3 Background 9
  3.1 Structure-Based Virtual Screening .......................... 9
  3.2 Big Data analytics ........................................... 10
  3.3 MapReduce .................................................. 11
    3.3.1 Consensus problem in MapReduce ................. 11
    3.3.2 MR open source ecosystem ......................... 13
    3.3.3 MR limitations .................................. 13
  3.4 Apache Hadoop ............................................ 14
    3.4.1 HDFS: Hadoop Distributed File System .......... 14
    3.4.2 YARN: Yet Another Resource Negotiator ....... 14
  3.5 Spark ..................................................... 15
    3.5.1 RDD: Resilient Distributed Dataset ............... 15
    3.5.2 Shared variables .................................. 16
    3.5.3 Consensus problem in Spark ...................... 16
    3.5.4 Spark SQL and Spark MLlib ....................... 17
    3.5.5 Why Spark for SBVS ............................... 18
  3.6 Cheminformatics .......................................... 18
    3.6.1 Molecular representations: SMILES and SDF .... 18
    3.6.2 Molecular libraries: ZINC and eMolecules ....... 19
    3.6.3 OpenEye toolkits ................................ 19

4 Implementation 21
  4.1 Programming model ...................................... 21
  4.2 Implementation details ................................. 22

5 Examples 23
  5.1 Standardization pipeline ............................... 23
  5.2 Screening pipeline .................................... 24

6 Materials and methods 26

7 Results 28

8 Discussion and conclusion 31

References 32
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Application Manager</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming interface</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DN</td>
<td>Data Node</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>HTS</td>
<td>High-throughput screening</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MR</td>
<td>Map Reduce</td>
</tr>
<tr>
<td>NGS</td>
<td>Next-Generation Sequencing</td>
</tr>
<tr>
<td>NM</td>
<td>Node Manager</td>
</tr>
<tr>
<td>NN</td>
<td>Name Node</td>
</tr>
<tr>
<td>PDB</td>
<td>Protein Data Bank</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RDD</td>
<td>Resilient Distributed Data Set</td>
</tr>
<tr>
<td>RM</td>
<td>Resource Manager</td>
</tr>
<tr>
<td>SBVS</td>
<td>Structure-Based Virtual Screening</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Manager</td>
</tr>
</tbody>
</table>
1 Introduction

High-throughput methods in structural biology allowed to produce massive virtual molecular libraries in the past decades. ZINC [1] and eMolecules [2] are two well-known examples of that. Since these libraries contain millions of molecules, using them in structure-based virtual screening (SBVS) is computationally heavy. However, SBVS is a trivially parallelizable task. Many tools such as Multilevel Parallel AutoDock4.2 [3] and OEDocking [4] use Message Passing Interface (MPI) [5] in order to parallelize SBVS. Using MPI has some major disadvantages. First, organizations that wish to run MPI based SBVS need to reserve some running time on an high performance computing (HPC) cluster. This is in general more expensive than buying cloud resources. Furthermore, MPI does not offer much more than a message passing Application Programming Interface (API). This means that problems like data distribution, locality-aware scheduling, load balance, fault tolerance and scalability must be handled by the developers.

Big data analytics on inexpensive hardware was pioneered by Google’s MapReduce (MR) [6]. In MR-programming model, data distribution, locality-aware scheduling, load balance, hardware faults and scalability are handled by the MR framework. This makes MR applications able to run in the cloud, where virtual machines (VMs) are more likely to fail and network communication is slow. Even though Google’s MR-implementation is legacy, many other implementations are available in the open source ecosystem. Apache Hadoop [7] is with no doubt the most used open source MR-implementation. Even if Hadoop offers all of the features of Google’s MR, it has some disadvantages. First, there is no support for workflows in Hadoop. This means that in order to set up a pipeline some third party tools such as Spotify Luigi [8], or Apache Pig [9] need to be used. In addition, even if newer versions support dataset caching for iterative tasks, Hadoop still lacks broadcast variables support.

Spark is a cluster-computing framework that implements MR, allows in-memory iterative tasks and has built-in workflow support [10]. Furthermore, Spark offers several additional features such as global variables, global sort, machine learning, and more. We believe that MR and in particular Spark for SBVS are more flexible and scalable solutions if compared to MPI implementations. Furthermore, an in-cloud solution is particularly appealing to those organizations that can not access HPC resources, or that just want to try out SBVS and later scale to high-throughput molecular libraries.
2 Aims

The core of this project consists in the development and evaluation of a tool for massively parallel SBVS. The tool we developed is based on Spark, and therefore it benefits of all of the properties we discussed in the introduction: scalability, fault tolerance and cloud runnability. Different molecular libraries require different standardization protocols, and SBVS applies to a variety of use cases. Therefore, instead of offering a command line interface (CLI), we offer a high-level Scala [11] library that allows the user to define ad-hoc pipelines for its own cases. Figure 1 shows an example of usage.

```scala
val res = new SBVSPipeline(sc) //sc is the SparkContext
  .readSmilesFile("/path/to/smiles-lib.smi")
  .filter(OEFilterType.Lead) //lead-like filter
  .generateConformers(0,1) //generate 1 conformer per SMILES
  .saveAsTextFile("/path/to/output/leadlike-conformers.sdf")
  .dock("/path/to/receptor.oeb", OEDockMethod.Chemgauss4,
    OESearchResolution.Standard)
  .sortByScore
  .getMolecules.take(10) //take top 10 poses
```

Figure 1: Sample SBVS pipeline using the SBVSPipeline object. This pipeline takes a SMILES library as input. First the input is reduced to lead-like molecules and 3D conformers are generated. Then the resulting conformers are docked to a receptor and the top scoring molecules are returned. In addition, the intermediate lead-like conformers data set is saved to be reused in the future.

For the moment, please take figure 1 as an assay of the capabilities of the tool we developed, further details will be given in the following sections. In order to evaluate our tool we run a couple of real world cases using ZINC and eMolecules molecular libraries, along with some additional experiments that we designed in order to produce the speedup plots. Doing so we aim to show that Spark can be used to screen high-throughput molecular libraries, that it scales, and more in general that it is an effective solution for massively parallel pipelining.
3 Background

3.1 Structure-Based Virtual Screening

In drug development, organizations often test the interaction of leads to certain targets for safety reasons. Companies such as AstraZeneca, GlaxoSmithKline, Novartis and Pfizer include this stage in their drug development cycle, and typically perform this test \textit{in vitro} against several targets [12]. In order to screen a lead against high-throughput molecular libraries wet-lab high-throughput screening (HTS) is often used [13]. Nevertheless, HTS is expensive and it has low hit rate. On the other hand, \textit{in silico} methods are often cheaper and faster.

Structure-Based Virtual Screening (SBVS) is an \textit{in silico} method that has been successfully used to screen high-throughput molecular libraries [14, 15]. In this study we discuss SBVS based on molecular docking. Figure 2 shows a generalized SBVS pipeline.

![Generalized SBVS pipeline.](image)

A typical SBVS-workflow takes as input a molecular library and a target receptor. The molecular library needs to be preprocessed in order to filter undesired molecules and in order to generate 3D conformers. The molecular library preprocessing is a complex process that is subject to many parameters, some of which will be discussed in the following sections.

The target structure is usually retrieved from the Protein Data Bank (PDB) [16]. Therefore, often a preprocessing step is included in order to remove a ligand from the receptor and in order to convert the PDB-file into the docking software format.

In the docking step a molecular docking software is used in order to dock each molecule in the molecular library to the target receptor. This means that for each molecule the docking algorithm will return a pose that is supposed to fit the receptor pocket. Then, in the scoring phase a scoring function will give a
score to each pose. The higher the score the better the pose is supposed to fit the target receptor. Finally, in the post-processing phase a group of best-scoring poses is selected and returned.

For high-throughput molecular libraries the whole process is compute-intensive. Therefore, in such case the molecular library needs to be partitioned and processed in parallel. However, the post-processing phase needs to take into account the whole dataset, therefore some distributed computing techniques need to be used.

3.2 Big Data analytics

Big Data analytics is a novel challenge in computer science. A fascinating study carried out by Hilbert and Lopez [17] estimates how humankind data storage capacity grew from 1986 to 2007.

Figure 3: Humankind storage capacity from 1986 to 2007. Retrieved from [17].

Figure 3 shows a linear growth, and digital storage overcoming analog storage in 2000s. According to Hilbert and Lopez in 2007 the humankind was able to store $2.9 \times 10^{20}$ optimally compressed bytes. While nowadays we are able to store such a huge amount of data, we still have the problem of analyzing that in reasonable time. Big Data analytics refers to a family of methods aimed to address that problem.

High-throughput methods in life science account for the growth of data size of
biological interest. For instance, Next-Generation Sequencing (NGS) machines can sequence millions of reads in parallel, producing terabytes of raw data [18]. Therefore, nowadays Big Data analytics is taking momentum in life science as well. Also in SBVS, Big Data analytics applies, since we aim to screen high-throughput molecular libraries. In this study we showcased our tool with two well-known libraries: eMolecules [2] and ZINC [1]. The free version of eMolecules contains roughly 7 million molecules, while the whole ZINC database contains over 20 million molecules.

### 3.3 MapReduce

Google pioneered Big Data analytics with the Map Reduce (MR) programming model, and were able to analyze twenty petabytes of data per day [6]. MR comes with an implementation targeted to commodity computer clusters. This means, as we discussed in the introduction, that MR application are out of the box scalable, fault tolerant and cloud ready. In the MR-model, the programmer describe the computation with a Map and a Reduce function, and the MR framework takes care of implementation details such as locality-aware scheduling and data distribution.

The Map function takes a key/value pair in input and it returns a set of intermediate key/value pairs. In most MR-applications the input file contains single line records, and the MR framework passes each line to the Map function, where the input key consists in the line number and the input value consists in the line content. Usually the programmer uses the Map-function in order to cluster the input records.

The Reduce function takes a set of intermediate values with same key and returns a final key/value pair. In the Reduce facet, the programmer usually discerns something useful from each cluster generated in the Map functions. An example will make this more clear to the reader.

#### 3.3.1 Consensus problem in MapReduce

Let us consider the consensus problem. Given an alignment of several sequences, we want to find the consensus sequence, that is the sequence that has the most frequent residue in each position of the alignment. For simplicity we suppose we do not have any gap in the alignment, and that each sequence has the same length. An example follows:
Figure 4: Consensus of a four sequences alignment. The consensus is the sequence that has the most frequent residue in each position of the alignment.

If each input sequence is stored in a different line of a text file, our code will look something like the following:

```
function Map(lineNumber, sequence)
    S ← ∅
    for p ← 1 . . . length(sequence) do
        S ← S ∪ {(p, sequence[p])}
    end for
    return S
end function
```

Figure 5: Map function pseudocode for the consensus problem.

```
function Reduce(p, P)
    m ← MostFrequentResidue(P)
    return (p, m)
end function
```

Figure 6: Reduce function pseudocode for the consensus problem.

As we said before the Map function takes a key/value pair in input. Therefore, since the input file contains a different sequence in each line, in figure 5 the input key represents a line number and the input value represents the sequence in that line. The pseudocode in figure 5 clusters each residue in the input sequence position-wise. This is done by producing an intermediate key/value pair for each residue in the sequence, where the key contains the residue position and the value contains the residue itself. Then each position/residue pair is returned inside a set of intermediate pairs. For instance, if we call Map(1, ACCCT) then the return set will be \{(1,A),(2,C),(3,C),(4,C),(5,T)\}.

Figure 6 contains the pseudocode for the Reduce-function. In figure 6 the inter-
mediate key \( p \) is a position number, and the set of intermediate values \( P \) contains some residues at that position. If the Reduce-code is defined in order to be commutative and associative, we can assume that \( P \) contains all of the residues at position \( p \). Therefore, in figure 6 the most frequent residue \( m \) is extracted, and a final key/value pair \((p,m)\) is returned. For example \( \text{Map}(1, \{A,C,C,T\}) \) returns \((1, C)\). Finally, the MR-framework is responsible for collecting all of the final key/value pairs and producing the consensus sequence.

### 3.3.2 MR open source ecosystem

In this section we want to give a brief overview of the frameworks available in the opensource MR ecosystem, other than Spark.

Apache Hadoop is with no doubt the most used open source MR implementation. In recent versions Hadoop can run many type of distributed applications other than MR, and it supports dataset caching for sequential MR operations. Many distributed frameworks such as Apache Pig [9], Apache Hive [19], Apache Flink [20] and even Spark [10] take advantage of this. Pig and Hive provide a layer of abstraction on top of Hadoop. Pig is script oriented, and therefore more appealing to scripting developers, while Hive is database oriented and more appealing to SQL developers. Flink is a young project that is somewhat comparable to Spark, and it has a sophisticated support for pipelining.

Another notable tool in the MR ecosystem is Spotify Luigi [8]. Luigi offers a level of abstraction that allows the user to pipe dataset transformations written using different frameworks (such as Hadoop and Spark). In addition, Luigi can schedule independent jobs on different machines, and it has a nice User Interface (UI) to monitor the overall pipeline status. Finally, MongoDB [21] is a document oriented Database Management System (DBMS) that has built-in support to MR transformations.

### 3.3.3 MR limitations

Even though Google’s MR pioneered Big Data analytics, it has some limitations that make it not suitable for certain problems. First it has a strict life cycle. In fact, in MR a Map dataset transformation have to be followed by a Reduce transformation. Furthermore, the MR specification does not say anything about shared variables such as broadcast variables or accumulators. Also, third party softwares for pipelining need to be used, and there is no caching of the dataset for iterative jobs. Finally, in MR there is poor support for global sorting.
3.4 Apache Hadoop

Apache Hadoop is the most used opensource MR framework. Only in 2008 Facebook already had multiple Hadoop dedicated clusters with up to 2500 cores each \[22\]. Furthermore, in 2008 Facebook was loading 250 gigabytes of compressed data per day in their Hadoop clusters, running hundred of jobs per day. Other well-known organizations that use Hadoop are Spotify, Netflix, eBay, Last.fm, LinkedIn and Yahoo \[23–28\].

While in the beginning Hadoop only provided a plain MR implementation, it evolved in order to support a variety of distributed applications. Hadoop makes this possible exposing the API of two of its main components: the Hadoop Distributed File System (HDFS) \[29\] and the Yet Another Resource Negotiator (YARN) \[30\].

3.4.1 HDFS: Hadoop Distributed File System

As Dean and Ghemawat argue in the original Google paper \[6\], MR applications are supposed to run on inexpensive unreliable hardware, therefore a distribute file system is needed in order to provide file availability and reliability. HDFS is a service that implements a Big Data oriented distribute file system, and it is responsible for the availability of the data in the whole Hadoop cluster.

Since in Big Data analytics the files are expected to take several gigabytes of disk space the HDFS block size is considerably bigger than in general purpose file systems. Furthermore, in HDFS each block is replicated in different machines in order to allow a certain number of machine faults. In the default configuration the block size is 256 megabytes and each block is replicated in three different machines.

The HDFS architecture has two main entities: NameNode (NN) and DataNode (DN). The NN tracks the file system tree, and in which DN each HDFS block is stored. Then, the DNs are responsible for storing the blocks and sending periodical health signals to the NN. Since the NN is a single point of failure, one or more NNs typically run on separate machines.

3.4.2 YARN: Yet Another Resource Negotiator

YARN is a service that allocates and negotiates resources for a variety of distributed applications that can potentially run on a cluster. In Hadoop the MR applications run on top of YARN, and also other kind of applications can coexist within the same YARN cluster. For example we could have a bunch of Hadoop MR jobs running along with some other Spark jobs.

In YARN terminology, a container incorporates several resource types such as number of cores, memory allocation etc. The ResourceManager (RM) is a
component that accepts application submissions and negotiates containers for the whole cluster. When an application is submitted the RM is responsible for negotiating the ApplicationMaster (AM) container, and for restarting it in case of failure. In many setups, one or more RMs run on dedicated machines along with HDFS NNs.

The AM is a per-application service that negotiates/restarts containers with the RM for its own application. Furthermore, the AM attempts to get containers where the data relevant to the application is stored, minimizing the network communication.

Finally, on each compute node in the cluster runs a service called NodeManager (NM). The NM manages the containers within its node, and it reports their availability and health status to the RM.

3.5 Spark

Spark is a cluster computing framework that aims to overcome the limitations of MR while still providing scalability, fault tolerance and cloud runnability. As we discussed before, MR penalizes performance in applications where a working set undergoes successive transformations. Improving the performance for this class of problems was the aim of the original Spark implementation [10]. To achieve this goal Spark introduces a dataset abstraction called Resilient Distributed Dataset (RDD) [31].

Even if it can run stand-alone, Spark gives the possibility to access HDFS and YARN services. From our experience, running Spark jobs on top of HDFS and YARN is more flexible and user-friendly.

3.5.1 RDD: Resilient Distributed Dataset

An RDD is a read-only collection of records partitioned through the nodes in the cluster. It can be created either from stored data (e.g. HDFS), or through a transformation of a previously created RDD. While in MR only map and reduce transformations are allowed, Spark offers a broader set of transforms that can be applied in any order. Some transformations that are offered to the user out of the box, apart form map and reduce, are: sortBy, groupBy, filter, union and intersection. For a more detailed RDD transformations list please refer to Zaharia et al. [31].

Spark exposes RDDs through a Scala, Java [32] or Python [33] API. Therefore, the user can pick its favorite programming language among the previous, in order to define one or more RDDs. However, from our experience we noticed that the Scala API is better maintained and it leads to better performance. This is not surprising since Spark is implemented in Scala.
Apart from transformations, the user can apply actions to RDDs that either return a value, or save the data in the file system. In addition, a cache method can be called on RDDs in order to cache the data for future transformations. This is the key feature that allows the performance improvement in successive working set transformations. Finally, Spark computes RDDs lazily in order to pipeline transformations, and each RDD maintains enough information on how it was derived, so that in case of fault a lost partition can be recomputed.

### 3.5.2 Shared variables

Along with RDDs the Spark API offers to the user the possibility to define two types of shared variables: broadcast variables and accumulators.

Broadcast variables are read-only objects that are shipped to each node. They can be used in order to efficiently ship a large object that is needed in each node.

Accumulators are variables that can be updated only through an associative operation, and that become readable at the end of the computation. They are often used in order to implement counters or sums.

Comparing with MR, shared variables make Spark suitable for a broader range of problems.

### 3.5.3 Consensus problem in Spark

In the following we solve the consensus problem in Spark under the same assumptions we used in the MapReduce section. We choose to use the Scala API for this example.

```scala
//Spark context initialization
val conf = new SparkConf()
  .setAppName("Consensus example")
  .setMaster("yarn-cluster")
val sc = new SparkContext(conf)
//Compute consensus
val rdd = sc.textFile("hdfs://path/to/sequences.txt") //read the input
val consensus = rdd
  .flatMap(seq => seq.zipWithIndex.map(_.swap)) //first transformation
  .groupByKey //second transformation
  .map((p,P) => mostFrequentResidue(P)) //third transformation
  .collect //action
println(consensus.mkString) //print the result
```

Figure 7: Spark code for the consensus problem.
In Figure 7 we first initialize Spark. The SparkConf object is used in order to specify an application name, and the spark master. The Spark master is a parameter that specifies if the application will be run in stand-alone mode or using YARN. In figure 7 we set the spark master in order to run in YARN mode (line 4).

The first RDD is created reading the input sequences file from HDFS (line 7), and a cascade of transformations is then applied in order to compute the consensus sequence. The first transformation uses some “syntactic sugar” in order to emit the intermediate key/value pairs, like we have done in figure 5. Since in Spark the map transformation does not allow to emit multiple key/value pairs we use a flatMap instead. Then the lambda expressions we pass to the flatMap, zips each residue in the input sequence with its position (or index), producing a residue/position pair. Then, since we want the position to be the key in the intermediate pairs, we swap residue and position.

The second transformation groups the intermediate key/values pairs by key. Since we set the residue position to be the key, Spark will cluster all of the residues at the same position together.

The third transformation maps each group produced in the previous one to the most frequent residues in it. The lambda expression passed to the map transform its equivalent to the pseudocode in figure 6.

Finally, an action it is applied to the last RDD in order to collect the most frequent residue at each position (line 12), and the consensus is printed out.

3.5.4 Spark SQL and Spark MLlib

In this section we give a short overview of two interesting projects that are distributed along with Spark. These projects enrich the Spark API and implement many useful algorithms for big data analytics.

Spark SQL [34] introduces a level of abstraction for working with structured data. It supports input data structured in different flavours e.g. Apache Hive, Parquet [35] files, JSON [36] files, as well as standard JDBC/ODBC [37, 38] database connections. Spark SQL’s main feature is to allow SQL [39] queries over RDDs.

Spark MLlib [40] is a machine learning library. It defines a labelled point RDD type that can be used in order to train a model with different kind of algorithms. Labeled point RDDs can be created from LIBSVM [41] files, from custom file formats, and through RDD transformations. Spark MLlib offers supervised and unsupervised learning, and includes well-known algorithms such as Lasso [42], Ridge [43], SVM [44], random forest [45] and k-means [46].
3.5.5 Why Spark for SBVS

In this paper we discuss the applicability of big data analytic techniques in SBVS. Therefore, we first investigated which of the MR frameworks in the MR open source ecosystem best fits the SBVS pipelines. Hadoop and Spark are both mature projects, and have been successfully used by various organizations. However, even if it is possible to implement SBVS in Hadoop [47], the plain MR-model in it hardly fits the problem. First, in Hadoop each map transformation needs to be followed by a reduce one. In the SBVS pipelines datasets do not need to be reduced until the post-processing phase is reached. It is true that each of the steps before the post-processing phase could be merged in a big map transformation, but this would lead to poor testability and configurability. In Spark there is not such problem since the RDD transformations can be defined in any order.

In addition, there is no support for pipelines in Hadoop. This would make it mandatory to use third party products in order to define pipelines. In contrast, Spark RDDs are processed lazily, making it easy to pipeline transformations using Scala, Java or Python syntax.

Hadoop API does not either offer broadcast variables. This makes it tricky to send the target receptor to each node in the docking phase. Sure, it would be possible to pack the receptor in the map code as a binary array, or even manually copy the receptor in each node. However, this would be inefficient and in more in general tedious. Spark API offers broadcast variables that are meant to efficiently send read-only data to each node, and therefore it gracefully solves the problem.

Finally, the sortBy transformations in Spark, allows to sort poses by score easily if we compare to the shuffle and sort phase in Hadoop.

To summarize, we chose Spark for SBVS because it allows configuration of the order of the dataset transformations, it gives built-in support for pipelines, it has broadcast variables, and because it offers a simple way to sort the dataset.

3.6 Cheminformatics

Cheminformatics is a family of in silico methods aimed to aid the process of drug development [48]. Those include chemical representations, chemical libraries, and tools for chemical processing. In this section we give an overview of the cheminformatics methods we used in this study.

3.6.1 Molecular representations: SMILES and SDF

SMILES and SDF are two chemical structure representation formats [49, 50]. While SMILES is lightweight-oriented and can be used in order to represent
stereochemistry and atom connections in a molecule, SDF is a considerably
heavier format and provides a way to represent atom coordinates as well. Since
when using the SMILES format, a molecule can be represented with a single
relatively short string, SMILES molecules can be quickly written down by an
experienced user. However, many applications require the 3D-representation
provided by the SDF format. Hence, a typical case is to first represent a molecule
using the SMILES format and then to go through a preprocessing phase in
order to generate the relative SDF representation. Nevertheless, SMILES can
be stereochemically ambiguous, and even if they specify stereochemistry, more
than one low energy conformer can correspond to a stereoisomer. Therefore,
this preprocessing phase typically produces more than one SDF representation
for a SMILES molecule.

3.6.2 Molecular libraries: ZINC and eMolecules

ZINC and eMolecules are two molecular libraries suitable for high-throughput
virtual screening [1, 2].

ZINC is freely accessible and contains over 20 million commercially available
molecules. Each molecule in ZINC is in ready-to-dock format, and can be
downloaded in SDF-format. For this reason, ZINC-subsets can be used in SBVS
skipping the preprocessing phase.

In contrast, eMolecule is a legacy molecular library and can be fully accessed
only after subscription. However, a free version of the database containing
roughly 7 million molecules can be downloaded for free in SMILES and 2D SDF
format. Since those formats do not provide a 3D-representation, a preprocessing
step is required in SBVS.

Both ZINC and eMolecules, along with the library, give a rather sophisticate
web-application that allows to search and purchase molecules.

3.6.3 OpenEye toolkits

The OpenEye toolkits are a set of cheminformatics and modelling programming
libraries [51]. Since they expose a Java API, they are suitable to work with
Spark. In this study we mainly used three of the tools included in the OpenEye
toolkits package: MolProp, Omega and OEDocking.

MolProp is a tool for molecular property calculation and filtering. In SBVS
filtering is a crucial step in the molecular library preprocessing phase. In fact,
we do not want to dock a target receptor to molecules that do not look like
leads. For this purpose, MolProp offers some default filter types as well as the
possibility to define a custom type. Among the ready to use filter types, the
lead-like filter is particularly interesting for our case. In addition, the Mol-
Prop filter includes a preprocessing step that removes salts and metals from the molecules.

Omega is a molecular 3D conformer generator. As we explained before, molecular libraries that do not provide 3D information need to be preprocessed in order to be docked to a receptor. Therefore, this is a step that might be included in the SBVS preprocessing phase of some molecular libraries. Since several conformers may correspond to a 2D representation within Omega, two parameters can be specified in order to control how many of them will be produced. First, we can set the number of stereocenters to be considered for stereochemically ambiguous molecules. This means that if we decide to consider $N$ stereocenters, $2^N$ stereoisomers will be generated. Then, for each of the stereoisomers previously produced we can decide the maximum number of low energy conformers to return.

OEDocking is a tool for molecular docking. It requires molecular 3D representation for the ligands, and it can read them from a SDF-file. In contrast, the target receptor must be represented in the OpenEye legacy format. However, this is not a big deal since OEDocking provides a handy tool that can create a target receptor file from a PDB entry. In addition, since PDB entries are often provided with a ligand, the pocket can be recognized automatically. OEDocking implements an exhaustive search algorithm in which the ligand is rotated inside the pocket with a certain resolution. Then, each of the resulting poses is scored with a user specified scoring function, and only the one with highest score is returned.
4 Implementation

In this project we used Spark in order to parallelize the code. Even if, Spark-based applications are relatively simple to implement, some solutions that we used are not trivial and are discussed in this section. First, we discuss the programming model that we developed in order to make pipeline definition easier. Then, we give some implementation details.

4.1 Programming model

In order to provide better configurability we implemented a scala library that can be used in order to define ad-hoc SBVS pipelines. `SBVSPipeline` is the core class in our tool. When the user creates a new `SBVSPipeline` object, he/she specifies a Spark context, and we say that the pipeline is in the Init state. Then, each time the user calls a method from the `SBVSPipeline` object, a transformation in the underlying RDD occurs, and there might be a state transition. The graph in figure 8 shows states and transitions a `SBVSPipeline` object can undergo.

![Figure 8: SBVSPipeline object states and transitions graph.](image)

Within the Init state the user can call three methods: `readSmilesFile`, `readConformerFile` and `readPoseFile`. These read a user specified text file and changes the state respectively into SMILES, Conformer or Pose. The user is supposed to specify a SMILES-file if he/she uses the `readSmilesFile`, and otherwise a SDF file. In addition, `readConformerFile` should be used to read conformers, and `readPoseFile` should be used to read scored poses. Since the files in SBVS are big there is no control for the correctness of the provided file.
The **SMILES** state has two main methods. The first one is *filter*, and it can be used in order to filter out undesired molecules from the dataset. We support OpenEye’s default and custom filters. The other method that can be used within the **SMILES** status is *generateConformers*. It takes two parameters that specify how many stereocenters to consider for stereochemically ambiguous molecules, and the maximum number of low energy conformers to return for each stereoisomer. This method causes a transition into the Pose status.

The *dock* method can be used within the **Conformer** status. It docks each of the conformers in the underlying RDD to a user specified target receptor. In addition, the user specifies the ligand rotation resolution and a scoring function to use. The *dock* method causes a transition into the **Pose** state.

Two main methods can be used within the **Pose** state. The *sortByScore* method sorts all of the molecules in the underlying RDD by score. In addition, since during the conformer generation many conformers can be generated from a single SMILES molecule, the *collapse* method can be used in order to reduce the poses relative to a single SMILES molecule to N best scoring poses. None of the methods that can be called within the **Pose** state cause transition.

Finally, the *saveAsTextFile* method can be called within **SMILES**, **Conformer** or **Pose** states in order to save the underlying RDD into the storage system. It does not cause any state transition.

### 4.2 Implementation details

The *filter*, the *generateConformers* and the *dock* methods are implemented using OpenEye toolkits.

OEDocking requires the receptor file to be in its legacy format, and does not provide a method to read it from HDFS. Therefore, we read the receptor once from the general purpose file system and then we use a Spark broadcast variable in order to make it available in each node.

We implemented two custom Hadoop record readers in order to allow our application to read SDF and SMILES molecules from HDFS. Since OpenEye objects require a considerable amount of memory we need to reuse them as much as possible. For this reason a *record size* parameter in our custom record readers can be set in order to decide how many molecules will be read and stored in a single record. The higher the *record size* is, the more the OpenEye objects will be reused. However, too high setting of *record size* could lead to load imbalances.

The *saveAsTextFile* and *sortByScore* method were implemented with Spark built-in RDD transformations. Also the *collapse* method it is easy to implement in Spark. In fact, it consists in a “*group by id*” transformation followed by a “*reduce by best score*” one.
5 Examples

In figure 1 we already gave an example of SBVSPipeline usage. The aim of this section is to give a couple of additional examples for real world cases.

5.1 Standardization pipeline

As we discussed before, molecular libraries usually need to undergo a preprocessing phase in order to be ready for docking. This process can be computationally heavy, but once it is done there is no need to repeat it. For this reason organizations that maintain their own high-throughput molecular libraries usually have a standardization pipeline aimed to preprocess new molecules before insertion. Needless to say, the standardization pipeline definition depends on the molecular library purpose. A simple standardization pipeline defined using SBVSPipeline follows.

```java
new SBVSPipeline(sc)
    .readSmilesFile("hdfs://path/to/new-molecules.smi")
    .filter("/path/to/filter-rules.txt")
    .generateConformers(0,1) //0 stereocenters, max 1 conformer
    .saveAsTextFile("hdfs://path/to/standardized.sdf")
```

Figure 9: Simple standardization pipeline.

In figure 9 we start by reading an input SMILES file from HDFS. Then, we filter the dataset using some custom filter rules defined in a `filter-rules.txt` file. Finally, we generate and save the 3D-conformers in SDF-format. Please notice that at line 4 we specified maximum one low energy conformer per stereoisomer, and we assumed that every SMILES molecule in the input file specifies stereochemistry. The last one is generally a too strong assumption. Furthermore, it is good practice to have some primary custom filtering rules to apply to every dataset, and then use the default OEDocking filters as a refinement for more specific cases.

```java
new SBVSPipeline(sc)
    .readSmilesFile("hdfs://path/to/new-molecules.smi")
    .filter("/path/to/primary-rules.txt")
    .filter(OEFilterType.Lead)
    .repartition
    .generateConformers(2,1) //2 stereocenters, max 1 conformer
    .saveAsTextFile("hdfs://path/to/standardized.sdf")
```

Figure 10: “Ready-to-dock” standardization pipeline.
In figure 10 we defined a standardization pipeline that generates ready-to-dock molecules from an input SMILES file in which some molecules might not specify stereochemistry. Here we set the maximum number of considered stereocenters for stereochemically ambiguous molecules to 2, like suggested by Irwin et al. [1]. Furthermore, we apply a primary general rules filter and then we further refine the dataset applying the OpenEye’s lead-like default filter.

5.2 Screening pipeline

Let us suppose we have a ready-to-dock molecular library, that we either standardized ourselves or we retrieved from a provider such as ZINC. A typical use case is to screen this molecular library against a target receptor. A pipeline for such purpose can be defined using the `SBVSPipeline` as follows.

```scala
val res = new SBVSPipeline(sc)
  .readConformerFile("hdfs://path/to/conformers.sdf")
  .dock("/path/to/receptor.oeb", OEDockMethod.Chemgauss4,
    OESearchResolution.Standard)
  .collapse(3) //keep only 3 best scoring pose with same ID
  .sortByScore
  .getMolecules
  .take(30) //take first 30
```

Figure 11: Screening pipeline.

In figure 11 we start by reading an SDF conformers file from HDFS. Then we proceed with the docking phase, using Chemgauss4 [52] scoring function and Standard rotation resolution. An important assumption that we make is that in the standardization phase we set an identifier in order to recognize conformers that were derived from the same SMILES molecule. Hence, the collapse method is able to reduce the poses with the same identifier to only 3 best scoring poses. Finally, we sort the molecules by score and we take the top 30 hits.

Since the docking phase usually takes a lot of time, is important to checkpoint the computation after the docking phase. This consists in saving the poses back to HDFS before proceeding. This allows to postprocess the poses in a different way later without repeating the docking phase. Line 5 in figure 12 shows how to checkpoint.
val res = new SEVS PIPELINE(sc)
        .readConformerFile("hdfs://path/to/conformers.sdf")
        .dock("/path/to/receptor.oeb", OEDockMethod. Chemgauss4,
          OESearchResolution.Standard)
        .saveAsTextFile("/path/to/checkpoint-poses.sdf")
        .collapse(3) //keep only 3 best scoring pose with same ID
        .sortByScore
        .getMolecules
        .take(30) //take first 30

Figure 12: Screening pipeline with checkpoint.
Materials and methods

In order to evaluate the tool we run some tests in our private cloud. Therefore, we set up a Hadoop 2.5.2 cluster composed by twenty-seven Ubuntu 12.04 LTS VMs. Each VM had 8 Virtual Cores (VCores), 16GB of Random Access Memory (RAM), and 160GB of disk space. In the configuration, we call one of the 27 VMs master. The master runs YARN RM and HDFS NN services. Since the master is a single point of failure in our configuration, we do not run anything else in it. The remaining 26 slave VMs run YARN NM and HDFS DN services.

In the HDFS configuration the block size and replication is set to 256MB and 3 respectively. Therefore, this configuration allows in total ~4TB of storage capacity, 208 VCores and 416GB of memory for concurrent containers.

Figure 13 summarizes our Hadoop test environment.

In the environment we described above we run Spark 1.2.2 as YARN application.

First, we aim to evaluate the scalability of a couple of pipelines. In order to do so we computed the speedups for the simple standardization pipeline (fig. 9) and the screening pipeline (fig. 11). For simplicity, we do not consider the additional container in which the YARN AM runs. Therefore, we run the two pipelines using one single VCore container first. Then, we run the pipelines again using one 8 VCores container, two 8 VCores containers, up to five 8 VCores containers.

In addition, each container included 14GB of RAM. The input set contained 50 thousand SMILES-molecules for the simple standardization pipeline, and 2 thousand conformers for the screening pipeline. Both input sets where derived from random entries in the free version of eMolecules. The record size parameter for this scalability test was set to 10 for both pipelines.

Furthermore, we performed a more consistent test that mimics a real case. First, we used the pipeline in figure 10 in order to standardize 100 thousand random SMILES-representation from eMolecules. For the primary filter we specified the same set rules used in the ZINC preparation protocol [53]. Then, we mixed the resulting conformers to the HIV-1 protease target subset from ZINC, which
counts 60 conformers. Then, our aim was to use the pipeline in figure 12 in order to find HIV-1 protease hits in the mixed dataset. For this purpose, we generated the HIV-1 protease receptor file from the 1AJV PDB entry [54] using the OpenEye’s command line tool. This more consistent test was run using twenty-five 7 Vcores containers with 14GB of RAM, plus an additional single core container for the YARN AM. For this test record size was set to 80 for the standardization part and to 30 for the screening part.
7 Results

Running time tables and speedup plots for the scalability test follow.

Simple standardization pipeline

<table>
<thead>
<tr>
<th>VCores</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3h 55' 28&quot;</td>
</tr>
<tr>
<td>8</td>
<td>50' 01&quot;</td>
</tr>
<tr>
<td>16</td>
<td>27' 56&quot;</td>
</tr>
<tr>
<td>24</td>
<td>19' 43&quot;</td>
</tr>
<tr>
<td>32</td>
<td>15' 47&quot;</td>
</tr>
<tr>
<td>40</td>
<td>13' 43&quot;</td>
</tr>
</tbody>
</table>

Table 1: Running time table for the simple standardization pipeline (fig. 9) scalability test. Every row corresponds to a run over a 50 thousand SMILES-molecules input.

Screening pipeline

<table>
<thead>
<tr>
<th>VCores</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4h 15' 32&quot;</td>
</tr>
<tr>
<td>8</td>
<td>1h 12' 59&quot;</td>
</tr>
<tr>
<td>16</td>
<td>38' 25&quot;</td>
</tr>
<tr>
<td>24</td>
<td>29' 03&quot;</td>
</tr>
<tr>
<td>32</td>
<td>20' 07&quot;</td>
</tr>
<tr>
<td>40</td>
<td>17' 54&quot;</td>
</tr>
</tbody>
</table>

Table 2: Running time table for the screening pipeline (fig. 11) scalability test. Every row corresponds to a run over a 2 thousand SDF-conformers input.
Figure 14: Speedups plot for the *simple standardization pipeline* (fig. 9). Every bar corresponds to a run over a 50 thousands SMILES molecules input.

Figure 15: Speedups plot for the *screening pipeline* (fig. 11). Every bar corresponds to a run over a 2 thousands SDF conformers input.
In the methods section we described a more consistent run we performed over 175 VCores. In that run, the standardization of the 100 thousand SMILES-molecules took roughly 7 minutes and generated 40960 conformers. Therefore, the mixed dataset contained 41020 SDF-conformers. The screening of if against the HIV-1 protease took roughly 4 hours. Table 3 shows some information about the top 10 poses in the results.

### Top 10 hits

<table>
<thead>
<tr>
<th>Source library</th>
<th>Identifier</th>
<th>Popular name</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZINC</td>
<td>ZINC03944422</td>
<td>Ritonavir</td>
<td>2.90</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC28766263</td>
<td>Lopinavir Metabolite M-3/M-4</td>
<td>2.70</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC38158973</td>
<td>Acetyl-pestatin</td>
<td>2.22</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC28766266</td>
<td>Lopinavir Metabolite M-3/M-4</td>
<td>2.21</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC28766363</td>
<td>Lopinavir Metabolite M-1</td>
<td>1.70</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC03951740</td>
<td>Lopinavir</td>
<td>1.40</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC03914596</td>
<td>Invirase</td>
<td>-0.24</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC04544447</td>
<td>Fmoc-Lys(Z)-OH</td>
<td>-1.29</td>
</tr>
<tr>
<td>ZINC</td>
<td>ZINC03941496</td>
<td>Atazanavir</td>
<td>-2.11</td>
</tr>
<tr>
<td>eMolecules</td>
<td>11071606</td>
<td>n/a</td>
<td>-2.42</td>
</tr>
</tbody>
</table>

Table 3: Top 10 hits for the screening pipeline with checkpoint (fig. 12) run over the mixed dataset. The free version of eMolecules does not provide a popular name for the last hit.
8 Discussion and conclusion

The simple standardization pipeline (fig. 9) and the screening pipeline (fig. 11) count 5 and 8 lines of code respectively, and can be written in a few minutes by an experienced user. Therefore, the first advantage of our tool is productivity.

The results in table 2 and table 1 show how adding VCores the running time decreases considerably. In addition, the speedup plots in figure 14 and in figure 15 show a linear growth. This means that the two pipelines scale well. However, the speedup that we observe in the 40 core run is $\sim$17 in figure 14 and $\sim$15 in figure 15. In contrast, we would expect a speedup that approaches 40, that is the level of parallelism. This happens because some load imbalance occurs. We believe that the load imbalance can be attenuated tuning the record size parameter. Nevertheless, in molecular libraries some molecules are bigger than others and therefore the dataset splits will take different time to be processed. However, this is a problem that occurs also in MPI implementations.

The more consistent run shows how, with adequate cloud resources, real world cases can be run over night. Most of the molecules in table 3 are not surprisingly HIV-1 protease inhibitors from the ZINC subset. The eMolecules hit might be a lead HIV-1 protease drug, but since the free version of eMolecules does not provide enough information we cannot be sure.

The results shows that when using our tool a user can easily set up scalable massively parallel SBVS pipelines. However, many improvements in the tool, such as better load balancing, can be done.

In this study we have shown that Spark applies to SBVS, and it is in general a suitable solution for massively parallel pipelining. In addition, we point out that Big Data analytics is beneficial in life science datasets as well.

Finally, we want to emphasize that all our test were run in a cloud environment, and this open-ups SBVS to these organizations that do not own HPC facilities.
References


[34] Spark SQL. Version 1.3.0. The Apache Software Fundation, Forest Hill, MD, USA. http://spark.apache.org/sql/.


