Scaling Machine Learning for Target Prediction in Drug Discovery using Apache Spark

Dries Harnie*‡, Alexander E. Vapirev†‡, Jörg Kurt Wegner†, Andrey Gedich‖, Marvin Steijaert**, Roel Wuyts§‡¶ and Wolfgang De Meuter*
*Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium
†Janssen Pharmaceutica, Turnhoutseweg 30, B-2340 Beerse, Belgium
‡ExaScience Life Lab, Kapeldreef 75, B-3001 Leuven, Belgium
§IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
¶DistriNet, KU Leuven, B-3001 Leuven, Belgium
‖ARCADIA Inc, Rostra Business Center, 30 Zanevsky prospekt building 2A, 195112 St. Petersburg, Russia
**OpenAnalytics, Kortestraat 30A, 2220 Heist-op-den-Berg, Belgium

***Abstract*—In the context of drug discovery, a key problem is the identification of candidate molecules that affect proteins associated with diseases. Inside Janssen Pharmaceutica, the Chemogenomics project aims to derive new candidates from existing experiments through a set of machine learning predictor programs, written in single-node C++. These programs take a long time to run and are inherently parallel, but do not use multiple nodes. We show how we reimplemented the pipeline using Apache Spark, which enabled us to lift the existing programs to a multi-node cluster without making changes to the predictors. We have benchmarked our Spark pipeline against the original, which shows almost linear speedup up to 8 nodes. In addition, our pipeline generates fewer intermediate files while allowing easier checkpointing and monitoring.**

I. Introduction

The trajectory from a biological concept to a drug available to patients is expensive and typically spans over a decade. Drug discovery starts by mapping a disease to a scalable experiment in a test tube. This enables screening of libraries of chemicals for active compounds (hits), from which chemical starting points (leads) are selected. These leads are then taken through multiple follow-up assays in cellular systems and later animal models to optimize their potency on the intended protein targets implicated in disease, while controlling their activity on undesired targets associated with side effects. After the final step of fine-tuning delivery and dosing in test animals, the compound is transferred to drug development, where it is tested on human subjects in phases. The first phase assesses drug safety in healthy volunteers over the dose range considered. The second aims to establish proof-of-concept, ie. a tangible clinical benefit for patients. Finally, in the third phase, a drug label is defined, which specifies the exact disease indication, ie. patients eligible for treatment, and an optimized dosage regimen, and then quantifies the benefit in comparison to the pre-existing standard-of-care treatment. If approved, the drug label is disseminated to physicians. However, the vast majority of compounds that enter drug development do not make it through to approval.

To increase the odds of seeing the development of a drug through to the end, the drug discovery phase must identify compounds that score well on *efficacy* ("do they affect the desired protein?") and *selectivity* ("do they *only* affect the desired protein?"). A study performed on drug failures [1] revealed that in 2011–2012, 56% of failures in testing phase two or three were due to lack of efficacy and a further 28% due to safety issues. One way of determining these properties is by applying the compound to several kinds of cells in a lab environment and seeing whether they have the desired effects. This is a slow and costly process, however, especially as one protein target might be affected by thousands of compounds, and one compound might affect multiple targets.

An alternative is *virtual screening*: the identification of candidate compounds through computation [2]–[5]. This method takes advantage of the fact that compounds with a similar structure often interact similarly with the same proteins. Inside Janssen Pharmaceutica, the Chemogenomics project attempts to identify candidate compounds by deriving information from existing compound–protein databases by means of machine learning methods [3], [6], [7]. To this end, a number of "predictor" programs have been developed. These programs construct a model by training on known entries in the compounds–targets activation matrix. This model is then used to predict activation probabilities for unknown compounds–targets combinations.

For every predictor a number of parameters can be tuned, such as the distance and similarity metrics, certain thresholds, etc. Additionally, there exist different methods for generating fingerprints of compounds, which has an impact on the quality of the model learned [8]. In the search for the best model, this parameter space needs to be explored. Additionally, to prevent overfitting [9], cross-validation is used. This is a well-known technique where a part of the training data is withheld and used to gauge the accuracy of the trained model.

The predictors are implemented as a set of separate multi-threaded programs in C++, written using Boost and Intel Threading Building Blocks (TBB; [10]). Every predictor consists of a training stage, in which the program trains itself on the data in the training set, and a prediction stage where activations are predicted for the compounds. The prediction stage is implemented as a parallel pipeline in TBB, which enables the program to make use of all cores in a node. Unfortunately, the programs cannot easily be adapted to run on multiple nodes and business efforts are focused on improving the quality of prediction, not the speed.

In this paper, we show how we used Apache Spark [11] to lift the existing single-node pipeline to a pipeline usable on a multi-node cluster without modifying the original code. The result is that the current predictor programs can be applied in parallel, and newly developed predictors can enjoy the same benefits.

The rest of this paper is structured as follows: section II describes the goal of the Chemogenomics project and its current pipeline. Section III summarizes the problems we encountered with that pipeline. In section IV we show the core implementation of an equivalent pipeline using Apache Spark. Section V compares the two pipelines in terms of performance and the problems outlined earlier. Similar approaches to scaling external processes are discussed in section VI. Finally, section VII ends with concluding remarks and future directions for our Spark pipeline.

II. The Chemogenomics Pipeline

As mentioned in the introduction, the aim of the Chemogenomics pipeline is to identify candidate molecules that work on certain protein targets. In this section we describe the pipeline, its datasets and the parameter space that needs to be explored.

The activation rate of compounds (at a specific concentration) on targets can be plotted on a very large, very sparse matrix. The dataset used contains $330,000$ compounds and $3,000$ targets, with less than 0.01% of all activations known.

Figure 1 shows a schematic of what one run of the Chemogenomics pipeline looks like for a pre-selected set of training data and test data.

- The input to the pipeline consists of a compound database, which contains metadata about the compounds, and distinct sets of training and test data of the form $\langle \text{target}, \text{compound}, 0|1 \rangle$.
- The predictors[1] learn a model from the training data and calculate predictions for the entire targets $\times$ compounds matrix. Each entry in the matrix represents the probability of activation according to that predictor;
- Once all predictors have calculated their predictions, the predictions are combined ("fused", as in [7]) into one matrix.
- The output of the pipeline consists of this fused matrix, and a score which determines how accurate this matrix is when compared to the testing data.

Running the pipeline once is reasonably quick; however the goal is not just to generate the predictions, but to find the set of parameters that ensure the optimal model. To this end, every predictor will be executed a large number of times:

- As mentioned in the introduction, using different fingerprinting methods might enable better activity prediction. Each method can yield completely different results.
- For cross-validation purposes, the dataset used is split into n complementary training and testing sets. Each split will also yield different results.
- Different concentration levels of the compound might have different effects on the target (and others). To aid in finding candidate compounds that score well on selectivity, this must also be considered.
- The predictors used have one or two parameters each that can be tuned. In later stages of the project, stronger predictors such as SVM [13]–[15] will be used. These are not only slower than the current predictors, but have more parameters that can be tuned. Estimates put the parameter space for the stronger predictors at four orders of magnitude.
- Sampling is critical for deriving confidence intervals and applicability domains for teams. The number of required samples for maximum scientific value is estimated at 100.

A back-of-the-envelope calculation with five different fingerprinting methods, five folds and five concentration levels yields an estimate of $5 \times 5 \times 5 \times 10,000 \times 100 = 125,000,000$ runs of individual predictors.

Each of these runs generates (at an absolute minimum) $330,000 \times 3,000 \times 4$ bytes $\approx$ 4GB of predictions. The prediction fusion at the end of the pipeline merges the output of several predictors for the same parameters and input into one file. Even with this reduction, it is not feasible to store the output of more than a few runs. Fortunately the goal of the Chemogenomics project is to discover the optimal set of parameters, so the generated predictions can be discarded once their score has been determined.

III. Problems

We analyzed the individual programs in the Chemogenomics project and the pipeline itself. This yielded several "low-hanging fruit" optimizations. Going forward, we assume that subsequent optimizations will yield diminishing returns.

During initial attempts to lift the Chemogenomics pipeline, we discovered a number of problems:

Problem 1: Using multiple nodes
As stated, the existing pipeline does not support multiple nodes. Adapting the current code, even to something like Intel Concurrent Collections [16], requires significant reworking.

Problem 2: intermediate storage
As calculated in the previous section, the most compact storage for a full run of the testing dataset requires on the order of 4GB. For intermediate storage, however, the speed of generation is far more important. Additionally, calculating predictions for different compounds — given the same training data and parameters — requires that the different outputs can be merged easily. For these two reasons, the output typically contains $\langle \text{compound}, \text{probability} \rangle$ pairs, grouped by target. This significantly increases the space used by intermediate results. For example, using flat files to store prediction results from *one* predictor on the data set used in this paper requires 40GB of storage.

[1]The Parzen–Rosenblatt predictor and others are based on the predictors presented in [12], with as yet unpublished improvements made by a team at the University of Linz.

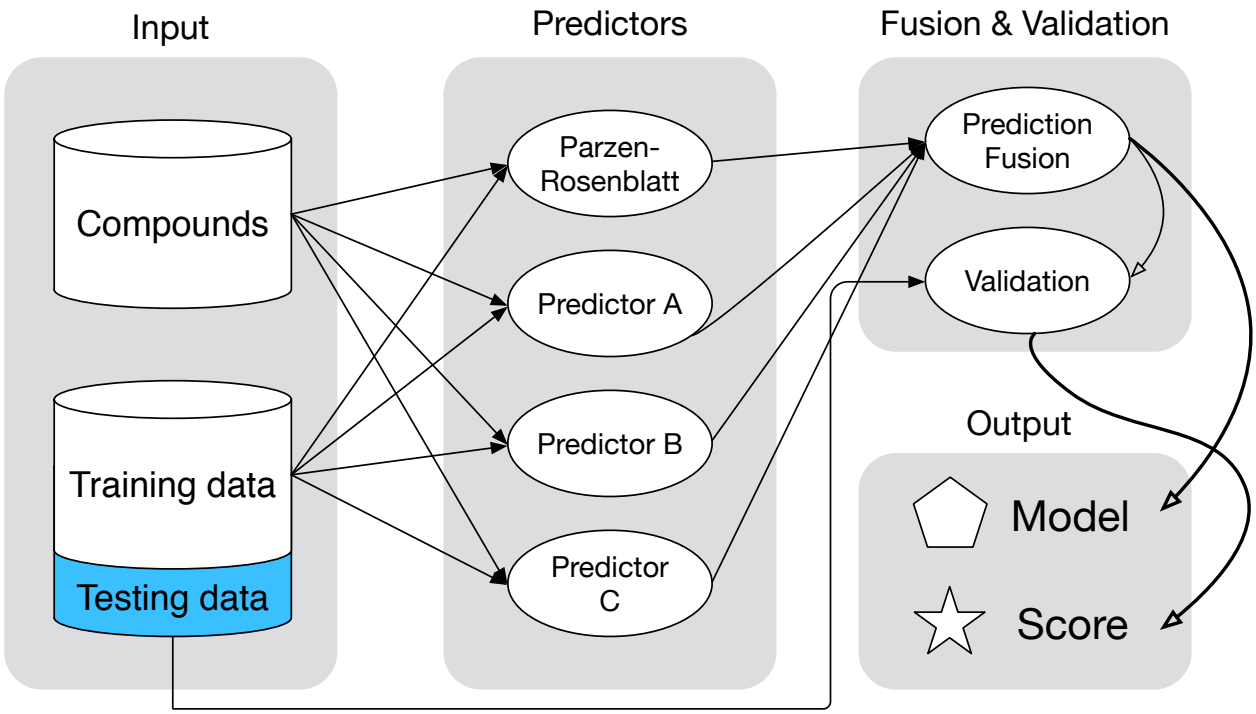

Fig. 1. Schematic of the Chemogenomics pipeline

Problem 3: Job management
Currently, every step of the pipeline happens on the same machine, in series. This is easy to implement and maintain, but there is no opportunity for checking the status of a run or recovering if a predictor or the entire node crashes.

Problem 4: Folds and sampling
Folds and cross-validation are critical instruments to improve the accuracy and scientific validity of the models produced. With the addition of sampling, hundreds of variations on the same dataset must be generated and stored. Distributing these variants to worker nodes puts a significant burden on the network as more nodes are added.

In the next section we discuss how our Chemogenomics pipeline, implemented using Spark, tackles these problems.

IV. Spark pipeline

This section first describes the Spark [11] programming model and runtime, and then shows how our Spark implementation lifted the Chemogenomics pipeline to multiple nodes. In addition, we evaluate various forms of intermediate storage.

A. Spark

Apache Spark consists of two parts: the first part is a programming model that creates a dependency graph. The second part is a runtime system which uses this graph to schedule work units on a cluster, and also transports code and data to worker nodes.

At the core of Spark programming model is the Resilient Distributed Dataset or RDD. An RDD is a homogenous collection of objects, combined with a partitioning function. This partitioning function assigns elements of the collection to partitions. RDDs can be constructed from native collections or from Cassandra, HBase, etc. RDDs can also be constructed from files on a local or remote filesystem like AmazonS3 or the Hadoop distributed filesystem [17]. Once constructed, the programmer can perform *transformations* (map, filter, etc) and *actions* (toArray, length, etc) on the RDD. Transformations are not performed immediately on the RDD; rather, they return a new RDD which depends on the old RDD. When the programmer performs an action on an RDD, all transformations are applied.

Spark also gives the programmer control over the storage of intermediate RDDs. By default, Spark is allowed to discard partitions and recompute them when memory is tight. If an RDD is expensive to recompute or if it is used several times, the programmer can indicate that the Spark runtime should persist it to a local cache. This caching ranges from in-memory caching (very fast but limited by memory) to on-disk caching (slow but plentiful). When a persisted RDD is no longer needed, the programmer can manually unpersist it or wait for it to be automatically freed when the Spark job ends.

A final note on the programming model is the transportation of values in Spark. Consider a transformation that looks up values in a shared `HashMap`. The programmer writes something akin to the following:

```
someRDD.map { x => ... map.get(x) ... }
```

where `x` is an element of `someRDD`, and the `get` method is used to look up the element in `map`. In order to execute this transformation on the various worker nodes, the code in braces must be accompanied by all external variables it references. In this case, for every partition of the RDD that is transformed, a copy of the map is sent along with the code. To solve

this problem, Spark provides *broadcast variables*, which are efficiently broadcast *once* to all worker nodes in the cluster. The value that is sent along with the code is then simply an identifier for the broadcast variable.

The Spark runtime is organized in stages, where every stage computes the result of an action applied to an RDD, and any transformations used to construct the RDD. Because RDDs are partitioned, transformations can be computed in parallel per partition. There are two kinds of transformations: "narrow" transformations — where each partition in the parent RDD is only used by at most one partition in the child RDD — and "wide" transformations, where a partition can be used multiple times. This distinction enables the Spark scheduler to exploit parallelism on a per-partition basis.

Spark is sometimes compared to the MapReduce [18] algorithm, of which Hadoop is the most popular implementation. For example, [19] states that MapReduce programs have a fixed topology as the name implies (map input data onto key–value pairs; group pairs by key; reduce values by key). By contrast, Spark programs can express more programs through various combinations of RDDs and transformations on them. For further comparison between MapReduce and Spark, we defer to [18].

Bux et al. [20] present an overview of scientific workflow management systems. Their overview is split along three dimensions: the kind of parallelism supported, the deployment options and the scheduling algorithm. Let us briefly evaluate Apache Spark among these three dimensions. First, Apache Spark supports data-, task- and pipeline parallelism using respectively partitioning, the stages in the scheduler and the merging of transformations in case of narrow dependencies. Secondly, Apache Spark can be deployed to either standalone nodes, or nodes in a cluster or cloud. Finally, Apache Spark is firmly in the "dynamic scheduling" camp, as the scheduler works in stages and supports recomputing partitions when needed.

B. Implementation

In this section we describe our Spark implementation of the Chemogenomics pipeline, titled S-CHEMO (pronounced like "eskimo"). As the name implies, it makes use of Apache Spark, using the Scala [21] programming language. Scala is a programming language that combines the object-oriented paradigm of Java with aspects from functional programming such as local type inference, functional combinators and immutable datastructures.

In order to launch S-CHEMO, a number of inputs and parameters must be provided. The first input is the path to the compound database, which was discussed earlier in section II. The second input is the path to the training data; if folds are required, these are generated on demand. Next, the predictors can be chosen along with extra parameters, both specific or common to all predictors. The predictor parameters are specified in a JSON file. Finally, the output format (see next subsection) and destination must be given.

Figure 2 shows the core of the pipeline. The parts of the pipeline that are not shown include configuration and parameterization code, as well as the specifics of how to invoke predictors. It consists of four parts, as indicated by vertical whitespace:

1) The database is queried for all compounds. An RDD is then created from these compounds, split up in partitions of `config.chunkSize` elements each. A smaller chunk size has the benefit of losing less work if a node crashes, but higher overhead as the Spark scheduler needs to create and monitor more jobs. Additionally, more jobs means more time spent merging at the end of the pipeline.
2) The `glom` transformation makes the partitions from the previous step explicit as `Arrays`, and the `zipWithIndex` transformation assigns each subsequent partition a sequence number. This number identifies the partition as it is written to disk and processed.
3) The third step is the most computationally- and IO-intensive. It is responsible for running all configured predictors (here in the `predictors` variable) on the given partition. This results in one slice of the targets $times$ compounds matrix for each of the predictors. These slices can be immediately fused together, discarding the intermediate one-predictor slices. The output of this step is thus an RDD of fused slices of the targets $times$ compounds matrix. Note the difference between the inner and the outer `map` calls: the outer one is called on an RDD so the Spark scheduler can perform it in parallel across multiple nodes. The inner `map`, meanwhile, is on a Scala sequence, so the `predict` calls are performed one by one on the same node.
4) The final step in the pipeline merges all fused slices to produce the final model. The `reduce` action requires its argument function to be *associative* and *commutative*, such that the Spark runtime can merge slices as jobs are finished. In our case, the `mergeFused` function copies non-overlapping slices of the targets $\times$ compounds matrix to the final output, so it is both associative and commutative. In the case of flat files (one of the file formats supported), simply concatenating output files suffices.

After submitting the job to the Spark cluster, the progress can be monitored using the built-in monitoring tools. Once the job is completed, the finished model can be retrieved from the output destination specified when the job was started.

C. Choice of output and intermediate formats

As part of the effort to lift the existing pipeline to multi-node clusters, different kinds of output formats were discussed with the development team at Janssen. When the code was originally developed, there were three requirements that influenced the choice of output formats: first of all, writing output files must be *fast*, as every predictor generates a lot of output. The second requirement was that multiple threads must be able to writing output *simultaneously*, since the predictor code is multi-threaded. The final requirement was that the format in question could be accessed from at least C/C++, Python and R (the latter two are primarily used for analysis). The addition of the Spark pipeline adds an additional requirement: it must

```
val compounds    = Source.fromFile(compoundsFile).getLines.toArray
val numChunks    = compounds.size / config.chunkSize
val compoundsRDD = sparkContext.parallelize(compounds, numChunks)
// compoundsRDD : RDD[MoleculeId] = [ABC, DEF, ...]

val partitionsRDD = compounds.glom.zipWithIndex
// partitionsRDD : RDD[(Array[MoleculeId], Long)] =
//  { ([ABC, DEF, ...], 1), ([XXX, YYY, ...], 2), ... }

val fusedRDD = partitionsRDD map { partition =>
  val predicted = predictors map { predictor => predictor.predict(partition) }
  // predicted : Array[(Predictor, Long, File)]
  return fusePrediction(predicted)
}
// fusedRDD : RDD[File]

val finalFused = fusedRDD reduce { (f1,f2) => mergeFused(f1,f2) }
```

Fig. 2. Implementation of the Chemogenomics pipeline in Apache Spark

be reasonably fast to merge the target $\times$ compounds slices of multiple nodes together to form the final matrix.

With these criteria in mind, the following output formats were considered:

Flat files were the default output format during the development and initial runs of the Chemogenomics pipeline inside Janssen. This output format generates one file per target, with lines that contain compound–activation probability pairs. This format satisfies all the original criteria. Since it is line-based, merging flat files is trivial.

HDF5 [22] is a hierarchical data format expressly designed for scientific applications. The main data type it supports is *a dataset*: a (multidimensional) array of values. These are organized in a tree, which can be traversed by means of a UNIX-style path. An early study done in-house showed that writing to HDF5 was substantially faster than writing to flat files. The official HDF5 library (`libhdf5`) supports C and C++, and bindings exist for Python and R. However, this library has one significant drawback: it is not thread-safe, especially with regards to writing[2]. We integrated HDF5 into the Janssen predictors using a global lock, effectively serializing writes. As expected, this change nullifies the advantage of HDF5 as only one thread can write to the file at the same time. There is also no ready-made solution to merge HDF5 files, but the data is grouped per target so probabilities can be appended reasonably easily.

Redis [23] is an in-memory key-value store, designed for very fast writes and reads. The keys are simple strings, while the values can be one of the following data structures: strings, lists, (sorted) sets and hash tables. In this case, the keys are the target names and their values are hash tables of compounds to probabilities. Unlike the previous output formats, Redis is actually a daemon that can be accessed via a UNIX or TCP/IP socket. The Redis protocol is very simple, so sending commands is very fast with little overhead. The Redis server can handle a very large number of concurrent connections, so the second criterium is satisfied. As with the other formats, Redis clients exist for all languages considered. Finally, pointing different nodes at the same Redis server is a trivial way to merge output.

In the Redis protocol, every command sent by the client results in a reply sent by the server. This turned out to be an issue — especially when accessing the Redis server over the network. We implemented a simple batching protocol, which resulted in a significant speedup.

A comparison of the throughput and output size of the various output formats can be seen in fig. 3. These timings were gathered by running the Parzen–Rosenblatt predictor on the same set of $3,000$ targets, generating probabilities for a set of $10,000$ compounds. Every timing reported is the median of fifteen runs, there were no significant outliers. For the first two formats, input was from files on local scratch storage and output was to local scratch storage. For Redis, measurements were made using a server running on `localhost`. The Redis server was completely flushed before every run. Note that all these formats group their output by target, each of which has a list of $\langle$compound ID, probability$\rangle$ pairs. The Compound ID is expressed as a 27-byte InChiKey, so every pair needs at least 31 bytes. The absolute minimum storage required for these formats is then $10,000 \times 3,000 \times 31$ bytes ≈ 930 MB.

As can be seen from the graph, flat files are the best file format with regards to both size and time spent creating them. Their biggest weakness is random access, but to analyze one target, the analysis scripts will load the entire file in memory first anyway.

V. Evaluation

In this section we compare S-Chemo against the original pipeline, both using the same binaries from the original Janssen pipeline. All test runs were made using the ExaScience cluster at the ExaScience Life Lab, Leuven, Belgium. This cluster consists of 30 nodes, each equipped with dual 6-core Intel® X5660 Xeon® CPU with 12 hardware threads each @ 2.80GHz and 96 GB of RAM. Communication between nodes was done using a standard gigabit network connection. The dataset used has $3,000$ targets and $330,000$ compounds as previously mentioned. The chunk size specified was $10,000$,

[2]http://www.hdfgroup.org/hdf5-quest.html#mthread

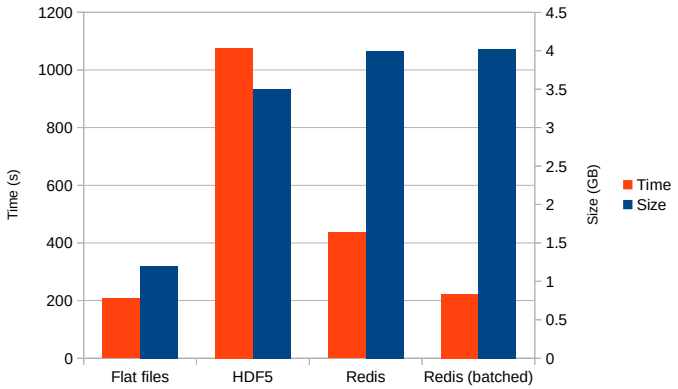

Fig. 3. Comparison of time taken for writing output (left bars) and the resulting size (right bars)

yielding 32 chunks in total. The remaining compounds are distributed evenly across chunks.

We present our results in fig. 4. The X axis lists the number of nodes allocated to computation, the Y axis represents the total time taken in minutes. The first two results were made by running the predictors in series in one node, then on three nodes using GNU Parallel [24]. These results are respectively the sum of all prediction times, and the longest prediction time. The rest of the graph depicts the time taken for running S-CHEMO on 1 through 8 nodes. As can be seen, the total time taken to calculate all probabilities decreases roughly linearly with the number of nodes. The speedup for S-CHEMO on up to twenty nodes is reported in fig. 5, and the values are shown in the appendix, in table I. The jumps at 11 and 16 nodes can be explained by the number of partitions (32) in the experiment, as fewer rounds are needed.

Note that S-CHEMO with two nodes outperforms the simple parallel case with two or three nodes: the reason is that one of the predictors takes a longer time to run during which the other two nodes are idle. By contrast, the "Spark-2" case always keeps both nodes busy because it works with smaller partitions.

Let us revisit the problems listed in section II.

Problem 1: Using multiple nodes
The Spark runtime automatically distributes work on a per-partition basis to nodes in the cluster. The programmer does not need to write any code for distribution or parallelism, only use the built-in transformations on RDDs.

Problem 2: intermediate storage
The use of intermediate storage is drastically reduced as compared to original pipeline, as all predictors are run in series on the same machine. The resulting fused predictor can immediately be validated, while other stages are still running. This frees up machines and disk space for use in other experiments.

Problem 3: Job management
The Spark pipeline hides a lot of job management and scheduling from the programmer. When a machine crashes, its jobs are recovered by the scheduler and retried. Crucially, the partitions of an RDD can be recreated at will, or recovered from persistent storage. Compare this to the manual approach where a system administrator needs to identify the jobs that were on the crashed machine and restart them manually. As for monitoring, both the "standalone" cluster that comes with Spark and more advanced deployment options enable system administrators to monitor running tasks.

Problem 4: Folds and sampling
Instead of generating the folds beforehand and distributing them to worker nodes, we can distribute the original dataset and have worker nodes create the folds on-demand. This saves network bandwidth. This is even more important for sampling, where only a few data points are removed and the rest of the dataset stays the same.

One other benefit of using Apache Spark to drive the Chemogenomics pipeline is the ease of development. Because the structure of the code dictates the structure of the execution across multiple machines, a small change can have a profound effect on execution. For example, say that we want to parallelize the execution of the predictors per partition as well. This involves a simple change to the code:

```
partitionsRDD.crossProduct(predictorsRDD).map {
  (part, pred) => ...
}
```

where `part` and `pred` are respectively a partition and a predictor object.

For all its benefits, the Spark programming model does have one downside: the programmer cannot indicate how many cores a given transformation or action uses. When the Spark runtime needs to compute the contents of an RDD, it creates one task per partition. The tasks are assigned to the nodes in the cluster which report idle cores. In our case, each task runs a set of predictor programs, each of which performs its work on as many cores as the hardware offers. Because the Spark scheduler is unaware of the cores claimed by the predictor programs, the Spark scheduler can assign jobs to cores in the cluster such that nodes are severely oversubscribed. (for our cluster, it has the potential to launch $24 \times 24 = 576$ simultaneous threads on one node!)

In order to prevent this, there are two options: restrict the predictor programs to a single core, or reduce the predictor programs active per node. The first can be realized with a "max cores" option supported by the predictor programs. By setting it to one, one predictor program is spawned per core. This is roughly comparable performance-wise: for four nodes it takes 78 minutes for a run instead of 72, but wastes potential work sharing in the multi-threaded predictor programs. The second option makes use of the `spark.task.cpus` property for the Spark program, which governs the number of cores to use per task. By default it is set to 1, but we can set this number to the number of cores per node, thereby claiming exactly one node per task. This solves the immediate problem, but potentially reduces the parallelism available for regular Spark transformations and actions.

VI. RELATED WORK

As Spark is a relatively new technology, there are only few publications about it. One such publication is [25], which shows and implements two large-scale classification algorithms

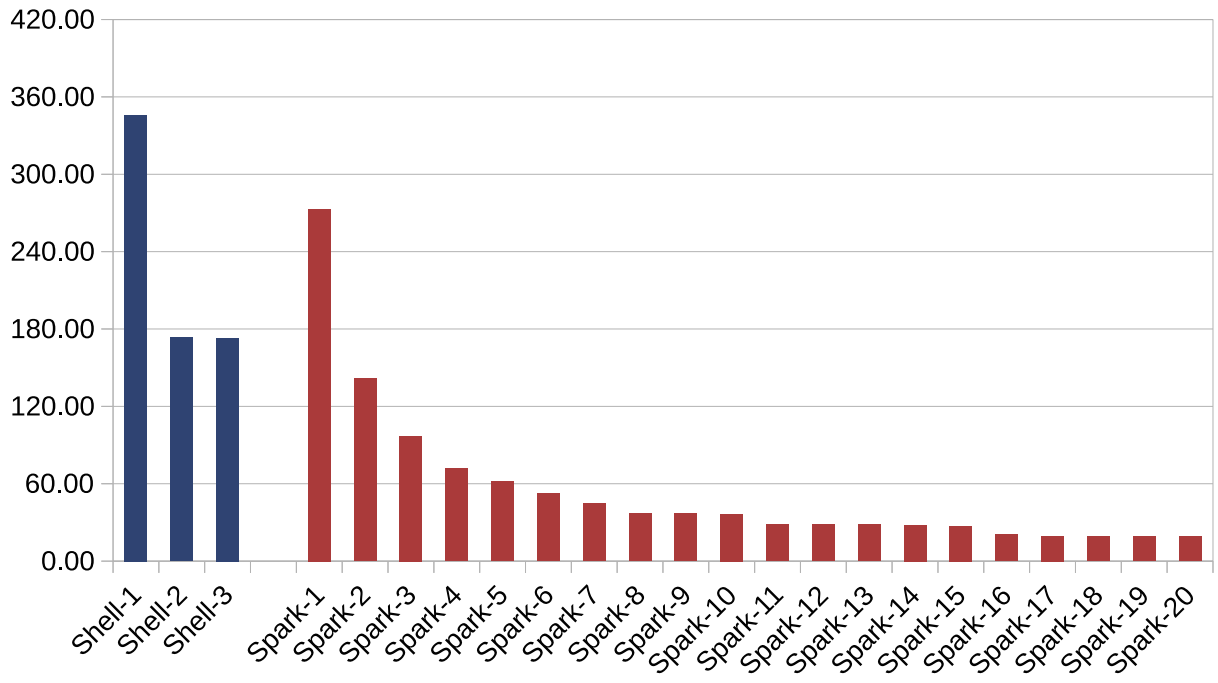

Fig. 4. Time taken to calculate probabilities (in minutes) vs. number of nodes, for regular execution (left) and S-CHEMO (right)

(logistic regression and a linear SVM) on top of Spark. The paper is also valuable for new developments using Spark: it explains how each Spark feature can be used to speed up their algorithm. Next, SparkSeq [26] is a platform for analysing sequencing data using Spark. The authors state that its main benefits are keeping the datasets in memory instead of on disk and that it enables interactive analysis. Finally, [27] is both an accessible introduction to Spark and an overview of Spark projects "in the wild".

One of the languages mentioned in the overview of Bux et al. [20] is the Swift language [28], which describes itself as "a scripting language designed for composing application programs into parallel applications that can be executed on multi-core processors, clusters, grids, clouds, and supercomputers". In Swift, the execution of an external program is expressed at the same level as function calls in ordinary programming languages. As long as the inputs and outputs of these programs are fully expressed in Swift, its interpreter can automatically parallelize and pipeline the workflows expressed. Swift also takes care of moving data to where computations are performed. In our scenario, this is actually a downside as we want to *avoid* moving the intermediate results around. Additionally, complex manipulations using Swift require a substantial amount of development, either in Swift or in an external program.

On a lower level, GNU Parallel [24] could be used to run the Chemogenomics pipeline on multiple machines. This tool can run commands in parallel on either a per-file or a per-record basis and retrieve the outputs. There is limited support for pipelining and scheduling, but developing and maintaining a pipeline using GNU Parallel is non-trivial. By contrast, Spark exploits parallelism and pipelining by the construction of the different RDDs. Moreover, it is embedded in a general-purpose programming language, so the programmer can better express himself.

VII. CONCLUSION

In this paper we discussed the Chemogenomics project within Janssen Pharmaceutica in the context of virtual screening, a more cost-effective way of discovering drugs. This project attempts to extrapolate candidate compounds that work on a given protein target from existing lab activation results determined in the lab. The existing pipeline was implemented as a set of separate programs which communicate using flat files. Changing these programs to work on a multi-node cluster was not possible, given that engineering resources were spent on new predictors.

In order to lift this pipeline to multiple nodes, we developed S-CHEMO, an implementation of the same pipeline using Apache Spark [11]. We showed benchmarks which proved that S-CHEMO scales linearly given the number of nodes. Our implementation partitions the prediction work among the various compounds and immediately produces the fused result. This means that intermediate data is immediately consumed again on the nodes that produce it, saving time and network bandwidth.

After the initial success in driving one run of the Chemogenomics pipeline using Apache Spark, we intend to lift it one step higher and also use it to drive the parameter optimization problems inherent to Chemogenomics. We are convinced that, using well-known techniques from machine learning, such a Spark-driven exploration of the parameter space will yield better results faster. This is especially important in the big parameter spaces connected to stronger but slower learners such as support vector machines [13]–[15] will be used. These

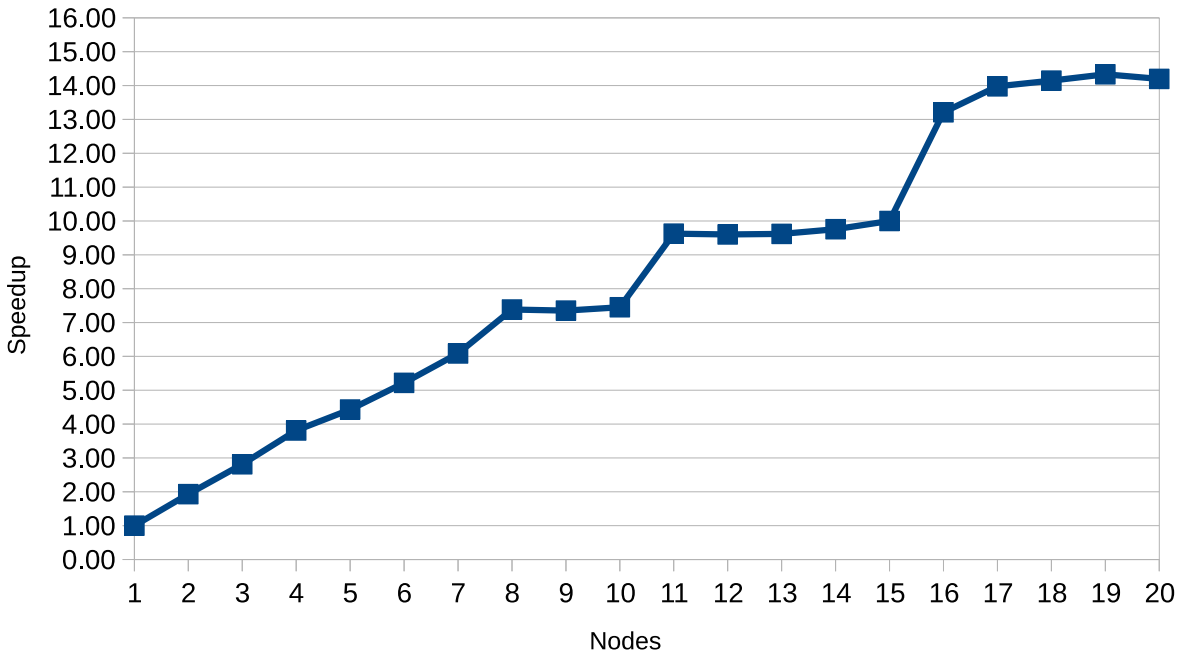

Fig. 5. Speedup results for Spark. We can observe almost linear speedup up to eight nodes and a 13x speedup for sixteen nodes.

stronger learners will also need more fine-grained partitioning and scheduling.

ACKNOWLEDGMENT

This work is funded by Intel, Janssen Pharmaceutica and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). All research was performed at the ExaScience Life Lab at IMEC, Leuven, Belgium. Sepp Hochreiter, Andreas Mayr and Gunther Klambauer at the University of Linz were responsible for various improvements to the predictor programs used in our benchmarks.

REFERENCES

[1] J. Arrowsmith and P. Miller, "Trial Watch: Phase II and Phase III attrition rates 2011-2012," *Nature Reviews Drug Discovery*, vol. 12, no. 8, pp. 569–569, Aug. 2013.

[2] K. H. Bleicher, H.-J. Böhm, K. Müller, and A. I. Alanine, "Hit and lead generation: beyond high-throughput screening," *Nature Reviews Drug Discovery*, vol. 2, p. 378, 2003.

[3] N. Wale, "Machine learning in drug discovery and development," *Drug Development Research*, vol. 72, no. 1, pp. 112–119, Feb. 2011.

[4] L. Wang, C. Ma, P. Wipf, H. Liu, W. Su, and X.-Q. Xie, "TargetHunter: An In Silico Target Identification Tool for Predicting Therapeutic Potential of Small Organic Molecules Based on Chemogenomic Database," *The AAPS journal*, vol. 15, no. 2, pp. 395–406, Apr. 2013.

[5] M. Schenone, V. Dančík, B. K. Wagner, and P. A. Clemons, "Target identification and mechanism of action in chemical biology and drug discovery," *Nature Chemical Biology*, vol. 9, no. 4, pp. 232–240, Apr. 2013.

[6] J. C. Costello, L. M. Heiser, E. Georgii, M. Gönen, M. P. Menden, N. J. Wang, M. Bansal, M. Ammad-ud din, P. Hintsanen, S. A. Khan, J.-P. Mpindi, O. Kallioniemi, A. Honkela, T. Aittokallio, K. Wennerberg, N. D. Community, J. J. Collins, D. Gallahan, D. Singer, J. Saez-Rodriguez, S. Kaski, J. W. Gray, and G. Stolovitzky, "A community effort to assess and improve drug sensitivity prediction algorithms," *Nature Biotechnology*, vol. 32, no. 12, pp. 1202–1212, Dec. 2014.

[7] G. M. Sastry, V. S. S. Inakollu, and W. Sherman, "Boosting Virtual Screening Enrichments with Data Fusion: Coalescing Hits from Two-Dimensional Fingerprints, Shape, and Docking," *Journal of Chemical Information and Modeling*, vol. 53, no. 7, pp. 1531–1542, Jul. 2013.

[8] S. S. Young, F. Yuan, and M. Zhu, "Chemical Descriptors Are More Important Than Learning Algorithms for Modelling," *Molecular Informatics*, vol. 31, no. 10, pp. 707–710, Oct. 2012.

[9] D. M. Hawkins, "The Problem of Overfitting," *Journal of Chemical Information and Modeling*, vol. 44, no. 1, pp. 1–12, Dec. 2003.

[10] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *NSDI '12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, Apr. 2012.

[12] A. Koutsoukas, R. Lowe, Y. KalantarMotamedi, H. Y. Mussa, W. Klaffke, J. B. O. Mitchell, R. C. Glen, and A. Bender, "In Silico Target Predictions: Defining a Benchmarking Data Set and Comparison of Performance of the Multiclass Naïve Bayes and Parzen-Rosenblatt Window," *Journal of Chemical Information and Modeling*, vol. 53, no. 8, pp. 1957–1966, Jul. 2013.

[13] X. Ning, H. Rangwala, and G. Karypis, "Multi-assay-based structure-activity relationship models: improving structure- activity relationship models by incorporating activity information from related targets," *Journal of Chemical Information and Modeling*, vol. 49, no. 11, pp. 2444–2456, 2009.

[14] L. Jacob and J.-P. Vert, "Protein-ligand interaction prediction: an improved chemogenomics approach," *Bioinformatics*, vol. 24, no. 19, pp. 2149–2156, Oct. 2008.

[15] N. Wale and G. Karypis, "Target Fishing for Chemical Compounds Using Target-Ligand Activity Data and Ranking Based Methods." *Journal of Chemical Information and Modeling*, vol. 49, no. 10, pp. 2190–2201, 2009.

[16] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, and F. Schlimbach, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3, pp. 203–217, 2010.

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST '10: Proceedings of the 2010 IEEE*

26th Symposium on Mass Storage Systems and Technologies. IEEE, 2010, pp. 1–10.

[18] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[19] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: a resilient distributed graph system on Spark." in *GRADES '13: Proceedings of the First International Workshop on Graph Data Management Experiences and Systems*, 2013, p. 2.

[20] M. Bux and U. Leser, "Parallelization in Scientific Workflow Management Systems," *CoRR abs/1303.7195*, 2013.

[21] The Scala Programming Language. [Online]. Available: http://www.scala-lang.org

[22] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," in *SC '99: Proceedings of the 12th conference on Supercomputing*. Proceedings of Supercomputing, 1999.

[23] Redis Website. [Online]. Available: http://redis.io

[24] O. Tange, "GNU Parallel - The Command-Line Power Tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011.

[25] C.-Y. Lin, C.-H. Tsai, C.-P. Lee, and C.-J. Lin, "Large-scale logistic regression and linear support vector machines using Spark," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 519–528.

[26] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics*, p. btu343, 2014.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over Hadoop data with Spark," *USENIX; login*, vol. 37, no. 4, pp. 45–51, 2012.

[28] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. T. Foster, "Swift: A language for distributed parallel scripting." *Parallel Computing ()*, vol. 37, no. 9, pp. 633–652, 2011.

APPENDIX

BENCHMARK RESULTS

TABLE I. BENCHMARK RESULTS FOR UP TO 20 NODES

	Total time (m)	Speedup
Shell-1	345.88	1.00
Shell-2	173.40	1.99
Shell-3	172.90	2.00
Spark-1	273.00	1.00
Spark-2	141.57	1.93
Spark-3	97.10	2.81
Spark-4	71.65	3.81
Spark-5	61.65	4.43
Spark-6	52.33	5.22
Spark-7	44.87	6.08
Spark-8	36.97	7.39
Spark-9	37.13	7.35
Spark-10	36.63	7.45
Spark-11	28.37	9.62
Spark-12	28.43	9.60
Spark-13	28.38	9.62
Spark-14	27.98	9.76
Spark-15	27.30	10.00
Spark-16	20.67	13.21
Spark-17	19.53	13.98
Spark-18	19.30	14.15
Spark-19	19.05	14.33
Spark-20	19.23	14.19