MaRe: Container-Based Parallel Computing with Data Locality

Marco Capuccini, Salman Toor
Division of Scientific Computing, Department of Information Technology, Uppsala University
Box 337
Uppsala, Sweden 751 05
marco.capuccini@it.uu.se
salman.toor@it.uu.se

Staffan Arvidsson, Ola Spjuth
Department of Pharmaceutical Biosciences, Uppsala University
Box 591
Uppsala, Sweden
staffan.arvidsson@farmbio.uu.se
ola.spjuth@farmbio.uu.se

1 Introduction

From genome analysis to high-energy physics, numerical analysis, drug design and more, software tools have played a major role in the scientific discovery process [11, 27, 33, 40]. While individual scientific tools are usually packaged by skilled developers, providing a robust research framework, real-world challenges require non-IT experts to combine and tune a variety of software components. As a result, the software “mashup” for certain studies is often unreproducible science [28]. Application containers tackle the issue by encapsulating complete software setups in a portable and self-describing manner, so that any compliant system can run with no additional dependencies, on any underlying computing infrastructure [32]. In fact, packaging container images is reasonably simple, and it provides a robust description for study-specific software setups, enabling researchers to reproduce scientific analysis without any laborious software configuration. Noticeable efforts in enabling container-based scientific analysis were made by the PhenoMeNal project (in medical metabolomics) [22], by the EXTraS project (in astrophysics) [19], and by the Square Kilometer Array project (in radio astronomy) [41]. A pattern that occurs in all of these research efforts consists in encapsulating computational tools in application containers and in enabling container pipelines through workflow systems. Hence, it is currently common practice to partition a scientific analysis into minimal and complete application containers, usually encapsulating disjoint software stacks, and then to run it as a pipeline of containerized tools using a workflow system (promoting container reusability). Unfortunately, this approach falls short when it comes to data-intensive applications, as most of the container-enabled workflow engines (Section 4) integrate poorly with the Big Data analytics ecosystems, and they do not provide locality-aware scheduling for parallel workloads. This last point is particularly penalizing. In fact, as Zhao et al. have shown, parallel containers that perform I/O-intensive operations against a decoupled storage system are subject to an intense bandwidth contention, thus causing a relevant performance deficiency [44]. This means that, unless an exotic hardware infrastructure is accessible, currently-available container parallelization methods perform poorly for data-intensive applications.

Data-intensive applications are becoming more and more prominent in science. Some well-known examples of this include: the 1000 genomes project (over 2.5K full DNA genomes) [17], the CERN's compact muon solenoid experiment (100GB/s throughput) [16] and the Square Kilometer Array project (over 400GB/s throughput) [41]. Google’s MapReduce (MR) is a cluster-computing model that has become ubiquitous both in industry and academia, as it is capable to process this magnitude of data on commodity computer systems [20, 25]. The two key ideas that MR introduces consist into: (1) saving network bandwidth through locality-aware scheduling and (2) automating the management of hardware failures. These features and many other challenging implementation details are seamlessly provided by an underlying runtime, which exposes an uncomplicated API to the developers. Even though MR is not publicly available, many implementations exist in the open source ecosystem. Among these Apache Spark, which represents an advancement over the original MR model, emerges as the Big Data cluster-computing framework that has collected the largest community [43]. Even though Apache Spark and similar frameworks provide transparent data locality and fault tolerance management, they are mainly designed to parallelize blocks of code and, to the best of our knowledge, none of these supports container-based processing.

Here we introduce MaRe, a programming library that extends Apache Spark, enabling container-based parallel computing with transparent support for data locality. Please notice that we stress data locality over fault tolerance, as the latter is already provided by publicly available workflow systems. MaRe supports Docker, the de facto standard application container platform [34], thus working with many of the existing container images. In addition, being based on Apache Spark, our system integrates seamlessly with the existing Big Data analytics ecosystem, and it inherits support for...
fault tolerance, in-memory processing and multiple storage systems. In summary, in this paper we make the following contributions:

- we propose an MR-like parallel programming model for container-based parallel processing (Section 2.1);
- we provide an associated implementation to our programming model, that enables seamless locality-aware scheduling for application containers (Section 2.2);
- we implement and evaluate two data-intensive scientific applications in MaRe, showing feasibility within our programming model and good scaling using inexpensive hardware resources (Section 3);
- we show MaRe integration with two storage backends from the Big Data ecosystem, with performance comparison (Section 3.2);
- we show container-based in-memory processing with MaRe, evaluating its performance (Section 3.3).

2 MaRe

2.1 Programming model

Being based on Apache Spark, MaRe has similar programming model. The control flow of the analysis is expressed by a driver program through a set of high-level primitives, which launch parallel operations over a cluster-wide dataset. The abstraction for the distributed data, and for the parallel operations, is provided by the Resilient Distributed Dataset (RDD) [42] programming interface. The vanilla RDD abstraction enables to process dataset records (and partitions) through user-defined lambda expressions, which are seamlessly parallelized by the underlying framework. MaRe builds on top of this mechanism, enabling to process RDD records via user-provided application containers (in addition to lambda expressions).

MaRe comes as a Scala [31] programming library that can be imported, as an add-on, into any Apache Spark application. A typical driver program starts by instantiating a MaRe object, which takes a string RDD in its constructor. In this way, the dataset can be loaded from numerous data sources (and optionally preprocessed), using the standard Apache Spark programming interface. MaRe objects are read-only (likewise RDDs), thus the control flow is defined by a sequence of parallel operations that are applied to successively- created instances. Data transformations are seamlessly applied to the underlying RDDs, which can be returned to the driver program at any point, thus results can be postprocessed and saved to any supported storage system via native Apache Spark code. Our programming interface is inspired by MR, and it provides four parallel operations though the MaRe object methods:

- map(imageName, command). It processes records using a user-provided command from a specified Docker image. The setInputMountPoint and setOutputMountPoint methods, from the MaRe object, can be called prior to this parallel operation, to configure where the input records will be made available in the containers, and where the command output is expected to be produced.

- reduce(imageName, command). It combines records using a user-provided command from a specified Docker image, returning a single value to the driver program. For the parallelization to succeed, the command should perform an associative and commutative operation over two dataset records. Prior to this parallel operation, the setReduceInputMountPoint1 and setReduceInputMountPoint2 methods, from the MaRe object, can be called to configure where the two input records are mounted in the containers. The command output path is configured as for the map parallel operation.

- mapPartitions(imageName, command). It processes records partitions using a user-provided command from a specified Docker image name. In contrast to the map operation, which feeds each record to containerized commands, this parallel operation mounts multiple records in each container (as a single file). Input and output mount points for the partitions are configured as for the map parallel operation.

- reducePartitions(imageName, command). Using a user-provided command from a specified Docker image name, it first combines records within each partition and then it combines the resulting partial results together, returning a single value to the driver program. In contrast to the reduce operation, which feeds two records to containerized commands, this parallel operation mounts multiple records in each container (as a single file). Likewise the reduce operation, the command should perform a commutative and associative operation. Input and output mount points for the partitions are configured as for the map parallel operation.

2.1.1 Example

Let us consider as an example an interesting yet simple problem in genetics. A DNA sequence can be represented as a text file written in a language of 4 characters: A,T,G,C. The GC content in a DNA sequence has interesting biological implications. For instance, there is evidence that GC-rich genes are expressed more efficiently than GC-poor genes [24]. Hence, within a large DNA sequence it can be interesting to count G and C occurrences. Given an Ubuntu Docker image [9], the task could be implemented as follows:

```scala
val rdd = sparkContext.textFile("sequence.dna")
val count = new MaRe(rdd)
.setInputMountPoint("/input.dna")
.setOutputMountPoint("/output.dna")
.setReduceInputMountPoint1("/input1.dna")
.setReduceInputMountPoint2("/input2.dna")
.map(
  imageName = "ubuntu",
  command = "grep -o '[GC]' /input.dna | " +
    "wc -l > /output.dna")
.reduce(
  imageName = "ubuntu",
  command = "cat /input1.dna /input2.dna | " +
    "awk '{s+=$1} END {print s}' > /output.dna")
```

Given a Spark context instance, which represents a connection to an existing Apache Spark cluster, we read the sequence using the textFile primitive, which returns an RDD. Then, we initialize MaRe passing the RDD sequence, and we set the input and output mount points for the parallel operations. By default Spark splits files in records line by line, hence we first use the map operation to count GC occurrences in each of them. In order to achieve this, as map command we use: (1) grep to filter GC occurrences from the input mount point, (2) wc to count each match and (3) standard output redirection to write the partial sum to the output mount point. This will emit a partial result for each RDD record, thus...
we use reduce to implement the global sum. Therefore, as reduce command we use: (1) cat to read the partial sums from the input mount points in a single string, (2) awk to sum the two records together and (3) standard output redirection to write the result to the output mount point. At job completion, the global sum will be returned to the driver program (in the count variable).

The previous code can be optimized by using mapPartitions and reducePartitions. In fact, in our first example we feed one or two records to each command, giving place to many unnecessary container initializations. This first approach is more suitable for time-consuming commands, where the processing time dominates over the container initialization time. In contrast, DNA file formats store genomes in relatively short lines, causing the container commands to exit almost immediately. Then, it is preferable to process whole partitions at once, considerably reducing the number of container initializations (and the relative overhead). Hence, we can modify the parallel operations as follows:

```scala
.mapPartitions(
    imageName = "ubuntu",
    command = "grep -o '[GC]' /input.dna | " + 
              "wc -l > /output.dna")
.reducePartitions(
    imageName = "ubuntu",
    command = 
              "awk '{s+=$1} END {print s} | awk '{s+=$1} END {print s}'} /input.dna " + 
              "> /output.dna")
```

Since mapPartitions and reducePartitions mount several records in each container through a multiline file, expecting the output to be in the same format, we may need to adapt the commands accordingly. For the map command there is no need for adaption, since grep can read and filter multiline files, as for the single line case. However, we can simplify the reduce command removing the cat merging part, as all of the input records are placed in a single file.

2.2 Implementation

MaRe comes as a thin layer on top of the RDD programming interface. The whole project counts less than 3K lines of code, and it relies on Apache Spark to provide data locality, as well as many other challenging features (e.g., fault tolerance, data ingestion, in-memory processing etc.). Most of the implementation effort consists of: (1) leveraging the existing RDD primitives to implement the parallel operations (Section 2.2.1) and (2) passing data between containers and RDD structures (Section 2.2.2).

2.2.1 Parallel operations

Figure 1a shows the execution diagram for the map parallel operation. The goal of this operation is to process an input RDD with a user-provided container image, producing a new RDD. The user-provided command expects to read a single record, thus we implement this operation using the flatMap RDD primitive, which enables to process records one-by-one through a provided lambda expression. Please notice that a similar logic can be implemented with the map primitive. However, by using the latter, the provided lambda expression can return only a single result (which is undesirable as containerized tools may produce multiple outputs). The lambda expression that we pass to the flatMap primitive starts a
Docker container feeding it the input record, it waits for the container to exit, and it returns the command output. The Apache Spark engine runs this lambda in parallel, with locality-aware scheduling, thus making sure that each container will process a record from a locally-available partition.

Figure 1b shows the execution diagram for the reduce parallel operation. The goal of this operation is to combine RDD records with a user-provided container image, returning a single value to the driver program. The user-provided command expects to read two input records to be combined, hence we implement this operation using the reduce RDD primitive. Our reduce lambda starts a Docker container feeding it the input records, it waits for the container to exit, and it returns the command output. The Apache Spark engine runs this lambda in parallel combining records two-by-two in each partition, and combining intermediate results together until there is a single intermediate result per partition. At this point, intermediate results are sent to the driver that combines them to a single value. Since each container command reduces the input in size, the network shuffle between workers and driver is inexpensive.

Figure 1c shows the execution diagram for the mapPartitions parallel operation. This operation has the same goal as map, however in this case we can assume that the user-provided command can read and output multiple records. Hence, we can mitigate the container-management overhead by processing whole partitions at once. Apache Spark enables this logic through the mapPartitions RDD primitive. In this case, our lambda starts a Docker container feeding it the input partition through a single file, it waits for the container to exit, and it returns the command output partition. As for the flatMap primitive, the Apache Spark engine handles partition locality for the scheduled containers.

Figure 1d shows the execution diagram for the reducePartitions parallel operation. This operation has the same goal as reduce, however here we assume that the user-provided command has the ability to read and output multiple records. As for the mapPartition implementation, we can mitigate the container-management overhead by processing whole partitions where possible. First, we apply the containerized command to each partition using the mapPartitions RDD primitive. In this phase our lambda expression starts a Docker container feeding it the input partition via single file, it waits for the container to exit, and it returns the command output partition. Since the user-provided command runs a reduce-like operation, combining all of the records, after this phase we obtain an intermediate result for each partition. At this point, we apply the reduce RDD primitive to combine intermediate results together, thus returning a single value to the driver. In this final phase, our lambda expression starts a Docker container feeding it the partial results, it waits for the container to exit, and it returns the command output. As in the previous cases, Apache Spark handles partition locality for the first processing phase, and little network overhead occurs in the final phase as reduce-like commands decrease the input in size.

2.2.2 Data handling

In-memory data processing is one of the advantages of the Apache Spark framework, over the original MR implementation (which penalizes iterative applications, such as machine learning). Hence, in order to retain performance, it is important to keep data in memory as it flows between RDD data structures and application containers. Since most scientific tools can only manipulate data through ordinary files, our implementation uses tmpfs [36], a memory-based file system, as an interface between RDDs and application containers. In fact, by using tmpfs for this purpose, we preserve Unix file semantics while keeping data in the operatve system page cache. Apart from retaining performance for iterative processing, this approach enables to perform concurrent containers reads and writes, without generating any disk I/O. This last remark has great importance when running multiple containers in the same worker node, as in this case disk I/O is not truly parallelized, causing a major resource contention [44].

Figure 2 shows how the data flows between RDDs and application containers. When an RDD record (or partition) needs to be processed by an application container, we first copy it in a user-provided tmpfs space, as a temporary file tmpfile. Then, we start a container mounting this file, along with an additional empty file tmpfile’ (which also lives in tmpfs). The containerized tool reads the data from tmpfile and it writes the results to tmpfile’, thus we can copy it in the resulting RDD’ record (or partition).

3 Evaluation

We evaluate MaRe on two data-intensive scientific applications: virtual drug screening (Section 3.2) and predictive toxicology (Section 3.3). The evaluation is performed both in terms of implementation feasibility, thus explaining how the use cases are implemented using the MaRe API, and in terms of scaling. We do not compare the results with other systems that do not provide data locality, as performance improvements when using locality-aware scheduling for data-intensive container-based operations were already shown by Zaho et al. [44].

3.1 Experimental settings

The evaluation experiments were carried out on the Uppmax region of the SNIC Science Cloud [39]. The region is built on top of discarded hardware, where physical nodes are based on dual CPU AMD Opteron 6220 processors, with up to 128GB of RAM and dual Gigabit Ethernet interconnection. In the cloud provider, we ran MaRe on top of a virtual Apache Spark cluster composed by 15 worker nodes, with 8 cores and 16GB of RAM each (providing a total parallelism of 120). As a storage backend both the Hadoop Distributed File System (HDFS) [35] and the SSC object store were
available. The latter is a fully managed solution, thus being decoupled from the worker nodes, however Apache Spark seamlessly ingests data splits in the worker nodes memory (preserving locality for subsequent computations).

### 3.2 Virtual drug screening

Virtual Drug Screening (VDS) is a computer-based method to identify potential drug candidates, by evaluating the binding affinity of virtual compounds against a biological target. Given a target 3D structure, usually representing a living-organism protein that we aim to interact with (thus altering its behaviour), in VDS we run a molecular docking software against a large library of known molecules. The docking software produces a pose, representing the orientation of the molecule in the target structure, and a binding affinity score. Hence, we rank molecules by score, and we return the top poses as potential drug candidates for the biological target. This process is both data-intensive, as libraries usually contain millions of compounds (e.g. ZINC [37]), and compute-intensive due to the molecular docking procedure (thus parallelization is necessary). A simple yet effective approach to VDS parallelization is to distribute the molecular library over several nodes, hence running the docking software in parallel. When the docking phase is completed, the top scoring molecules can be aggregated for the end user. This logic can be implemented in MaRe by mapping the docking software against the molecular library (abstracted by an RDD), and then by reducing the scored poses with a sorting tool that only returns the $N$ top elements. We note that by selecting the top $N$ poses, such a scoring tool performs a commutative and associative operation (enabling reduce-like parallelism).

We implemented VDS in MaRe using FRED [26] as molecular docking software and sdsorter [8] as poses sorting tool. Both software tools can read and output multiple molecules at the same time, hence we implemented the analysis using mapPartitions for the docking phase, and reducePartitions to aggregate the top-scoring molecules. This MaRe-based implementation counts ~40 lines of code. We benchmarked the implementation screening an HIV-1 protease representation [14] against the AK Scientific Inc. catalog [4], that we retrieved in ready-to-dock format from the corresponding ZINC library subset [10] (which counts ~1.2K molecules). The molecular library was made available both via HDFS and via object storage, for performance comparison. For both storage backends, we repeatedly run the benchmark using $20, 40, \ldots, 120$ cores. Furthermore, we ran the analysis using the software tools on a single-core SSC instance, thus we computed the absolute speedup and we ensured parallelization correctness by result comparison.

The analysis took 19.3 hours when running the tools on a single core, while the parallel implementation took 14 minutes using HDFS and 17 minutes using the object storage (on 120 cores). Hence, at full regimen we got an absolute speedup of 83 when using HDFS, and an absolute speedup of 68 when using the object storage. Figure 3a shows how the performance metrics evolve as the number of cores increases, for both storage backends. While speedup increases linearly when using the object storage, with HDFS it stalls between 60 and 100 cores, and then it considerably increases again at full resource regimen (giving best performance). This difference in the performance evolution reflects the distinct ingestion policies that are applied when using the two storage systems. In fact, while HDFS runs in the worker nodes, enabling full data locality, the object storage runs on a separated set of machines. Hence, in the first case Apache Spark assigns data splits to worker nodes a priori, trying to achieve full data locality, while in the second case it pulls data splits dynamically, as cores get free. Since HDFS does not fully replicate data splits in the cluster, when using a subset of the available cores, some splits become unavailable in the resource subset and network shuffling occurs. In this case, the object storage policy performs as good or even better than HDFS. In fact, while it is equally penalized by the initial network shuffling, this ingestion policy pulls splits dynamically incurring in fewer idle core time. In contrast, at full resource regimen complete data locality is achievable, thus the HDFS policy performs better as it avoids network shuffling.
3.3 Predictive toxicology

Given a complex system, with observable inputs and outcomes, predictive modelling aims to derive a statistical model from prior observations, with known outcome, in order to predict outcome of untested inputs. In toxicology, this method enables to derive statistical predictors for molecular toxicity [15]. Specifically, given a set of chemical compounds with known toxicity outcomes (namely training set) a machine learning algorithm, such as Support Vector Machines (SVM) [18], is applied to such observations to derive a predictive model. In addition, as toxicity predictors are aimed to enable risk assessment [29], it is crucial to assign confidence to predicted outcomes. Aggregated Conformal Prediction (ACP) is a mathematical framework that has been successfully adopted to tackle this challenge [29, 30, 38]. ACP requires to build several predictive models over different shuffles of the same training set, that are later aggregated in a single predictor. Each of the individual models retains a fraction of the training data as an external calibration set, which is used to assign object-specific confidence at prediction time. Hence, even if toxicology training sets are relatively small, the ACP model size increases with the number of aggregated predictors. As it is common practice to train over multiple training sets with parameter sweeping, the resulting model ensembles are large, making training and storing ACPs in a distributed environment desirable. This can be achieved by partitioning the building phase over a computer cluster, and by persisting each model locally to the node where it was computed. At prediction time, molecular representations with unknown toxicity outcome can be sent to the nodes, and the resulting confidence intervals can be computed by combining intermediate predictions from the distributed models. More in detail, for each observation with unknown outcome, the ACP algorithm computes the confidence intervals by taking the median predicted p-value from the individual models. Hence, the final combination can be performed by grouping results for each observation, and by computing the median for each group.

We developed distributed ACP-based predictive toxicology in MaRe, using CPSign [2]: a software tool for molecular modelling that leverages on conformal prediction and SVM. In this evaluation we trained and aggregated predictions from 100 models, as it is done in real-world scenarios [29, 30, 38]. We used data from a publicly available study [13] containing ~70K observations, of which we used 80% as training data and 20% as test data. The distributed model building phase is implemented using the map operation, against an RDD of 100 fixed seeds (for training set shuffling reproducibility). In the map command we ran the CPSign model building, specifying the seed and the path to the training data, which is available within the application container. Furthermore, since the CPSign output is in binary format, we encoded it using the base64 UNIX command (for string RDD compatibility). After this first phase, Apache Spark keeps the models in memory (with node locality), hence we applied a second map operation to perform the predictions. It is worth mentioning that we used map instead of mapPartitions in this phase, as CPSign cannot read multiple models from a single file. In the map command we decoded the model and we passed it to CPSign, to predict toxicity for the test data (that we made available within the application container). Finally, we combined the results from the distributed predictions, to compute the toxicity outcomes, using the standard Apache Spark API. This MaRe-based implementation counts less than 100 lines of code.

We ran the benchmark on 20, 40, … 100 cores (we did not run on 120, as we train 100 models in total), and on a single core SSC instance using CPSign (to compute absolute speedup and to ensure correctness by result comparison). We did not use any storage backend, as the molecular representations were wrapped in the Docker image along with CPSign, and because we stored the resulting models in the Apache Spark distributed memory. The analysis took ~4.5 hours when running CPSign on single core, while the parallel implementation took 6 minutes at full regimen, leading to an absolute speedup of 46. Figure 3b show how the performance metrics evolve as the number of cores increases. If we exclude the 80 cores run, the implementation scales linearly. Please notice that the speedup remains flat between 60 and 80 cores, because of a load balance issue. In the first case 60 models are build at first, and 40 models are computed in a subsequent phase (leaving 20 cores idle), while in the second case 80 models are computed at first leaving 20 models for the final phase (where 60 cores are idle). Hence, since each model takes roughly same time $T$ to be computed, the whole analysis takes $2T$ in both cases (leading to a flat speedup).

4 Related work

Container-based parallel computing is a fast-developing area. Here we focus on the computing engines that enabled analytics in prominent research initiatives. In all of the cases, these systems come primarily as workflow tools.

4.1 Systems that leverage on orchestration platforms

Container orchestration platforms provide key requisites for service deployments such as: scheduling, continuous deployment, high availability, fault tolerance, overlay networking, service discovery, monitoring and security assurance [23]. These platform are usually deployed in cloud environments, thus making computing power elastic and readily available with no upfront cost. The workflow systems that we discuss here mainly leverage on the cluster-scheduling capabilities of container orchestrators, hence using them as resource managers.

Luigi, Galaxy and Pachyderm are the workflow systems adopted by the PhenoMeNal project, for large-scale medical metabolomics [1, 3, 6, 22]. All of the three engines run on top of Kubernetes, the orchestration platform that has collected the biggest open source community [12]. While Luigi and Galaxy need a distributed POSIX-compliant file system, for container synchronization, Pachyderm uses its own shared file system (which is built on top of object storage).

Nextflow is another container-enabled workflow engine, that has seen many adoptions in the scientific domain [21]. The system can run on top of multiple cluster resource managers (including Kubernetes). Nextflow supports POSIX file systems, as well as the Amazon S3 object storage [5]. However, unless the containerized tools can read and write from Amazon S3, only a shared POSIX-compliant storage can be leveraged for for concurrent container synchronization.

All of these systems lack awareness of data locality. Hence, data is most likely accessed from a decoupled location, causing network contention for I/O-intensive operations. We note that remote access can also occur in MaRe, as we have shown in section 3. However, in this case the network shuffle occurs only initially, as Apache Spark operates intermediate results with locality awareness.
4.2 Systems that leverage on exotic hardware infrastructure

Here we describe workflow systems that leverage on hardware systems that provide low network latency. It is important to point out that under these settings network latency may be sufficiently low, achieving good performance without data locality.

Galaxy and Nextflow, two of the engines that we introduced in the previous section, can also distribute pipelines over high-performance computing systems (leveraging on publicly-available schedulers). DAVinci is another interesting system that was developed to process massive amount of information coming from the Square Kilometer Array (SKA) [41]. The execution framework is container enabled, and it leverages on many-tasks computing infrastructure to support both compute-intensive and data-intensive use cases. Unlike the systems that we previously introduced, DAVinci is not a general-purpose solution, and it is mostly designed around the SKA use case. Finally, locality-aware scheduling is currently under evaluation in DAVinci.

5 Discussion

MR-based processing pipelines have seen massive adoption in both industry and academia [25]. MaRe introduces container-based processing capabilities to these pipelines, opening some interesting opportunities. First, locality-aware scheduling (along with fault tolerance) enables for scaling container-based Big Data analytics on commodity computer clusters, as we have shown in the evaluation section. To the best of our knowledge this has not been achieved previously, as none of the available container parallelization systems provides support for data locality. Due to the increased adoption of application containers in the scientific domain, we believe this achievement to be particularly important. In fact, the degree of reproducibility that container-based pipelines can obtain is irrelevant if the analyses can only run on exotic hardware infrastructures.

Second, our MR-based programming model allows for effectively express distributed computation cpu-aware applications, without concerning about time-consuming distributed programming tasks. In fact, for the real-world use cases that we have shown, the complete analysis consists less than 100 lines of code. Even if we have shown two use cases, the success of the MR model over several domains suggests that MaRe will apply to many other scenarios.

Finally, implementing container-based analytics within the MR framework enables seamless interoperability with both established and cutting-edge Big Data technology. This contrasts with the systems that we highlighted in section 4, which have little or no support for Big Data platforms. In this study we have only shown support for multiple storage backends, and in-memory processing. However, being based on Apache Spark, MaRe inherits support for many other platforms, opening interesting opportunities such as container-based stream processing and interactive distributed processing.

6 Conclusions and future work

MaRe is a programming model for container-based data processing, implemented on top of Apache Spark. The system provides seamless locality-aware scheduling for distributed container operations, thus enabling container-based processing on inexpensive computing resources. In addition, being based on Apache Spark, our system integrates seamlessly with the Big Data open source ecosystem. These characteristics are missing in current systems for container-based processing, which mostly focus on workflow management. However, MaRe does not try to replace any of the available workflow engines, and it can be used as a pipeline step within these.

An interesting direction of future development of this work consists into evaluating MaRe for interactive and stream processing, as both features are provided by the underlying Apache Spark engine. MaRe is open source, and it is available on GitHub [7].

Acknowledgments

This research was supported by The European Commission’s Horizon 2020 programme under grant agreement number 654241 (PheNoMenAl). The computations were performed on resources provided by SNIC Science Cloud (SSC) under Project SNIC 2018/10-4.

References

[1] 2011–2018. Luigi. https://github.com/spotify/luigi (2011–2018). Accessed 06-04-2018.
[2] 2016. CPSign. http://cpsign-docs.genetassoft.com. (2016). Accessed 06-04-2018.
[3] 2017. Pachyderm. https://pachyderm.io. (2017). Accessed 06-04-2018.
[4] 2018. AK Scientific, Inc. https://aksci.com. (2018). Accessed 06-04-2018.
[5] 2018. Amazon S3. https://aws.amazon.com/s3/. (2018). Accessed 09-01-2018.
[6] 2018. Galaxy project. https://galaxyproject.org. (2018). Accessed 06-04-2018.
[7] 2018. MaRe. https://github.com/mcapuccini/MaRe. (2018). Accessed 09-01-2018.
[8] 2018. xdsoriter. https://sourceforge.net/projects/xdsoriter. (2018). Accessed 06-04-2018.
[9] 2018. Ubuntu Docker repository. https://hub.docker.com/_/ubuntu. (2018). Accessed 06-04-2018.
[10] 2018. ZINC AK Scientific subset. http://zinc.docking.org/catalogs/aksci. (2018). Accessed 06-04-2018.
[11] I. Antcheva, M. Ballintijn, B. Bellenti, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Malone, M. Goto, J. Iwaszkiewicz, A. Keshuk, D. Marcos Segura, R. Maunier, L. Moneta, A. Neumann, E. Offermann, V. Onuchin, S. Panacek, P. Rademakers, P. Russo, and M. Tadel. 2009. ROOT – A C++ framework for petabyte data storage, statistical analysis and visualization. Computer Physics Communications 180, 12 (2009), 2499 – 2512. 40 YEARS OF CPC. A celebratory issue focused on quality software for high performance, grid and novel computing architectures.
[12] Matt Assy. 2016. Why Kubernetes is winning the container war. http://www.infoworld.com/article/3118345/cloud-computing/why-kubernetes-is-winning-the-container-war.html. (Sept. 2016). Accessed 05-12-2017.
[13] Douglas S. Auld, Noel T. Southall, Ajit Jadhav, Ronald L. Johnson, David J. Diller, Anton Simeonov, Christopher P. Austin, and James Inglese. 2008. Characterization of Chemical Libraries for Luciferase Inhibitory Activity. Journal of Medicinal Chemistry 51, 8 (2008), 2372–2386.
[14] K. Backstro, S. Lowgren, K. Osterlund, J. Atepo, T. Unge, J. Tunge, J. Hulten, N. M. Bonham, W. Schaal, A. Karlen, and A. Hallberg. 1997. Unexpected binding mode of a cyclic sulfamide HIV-1 protease inhibitor. J. Med. Chem. 40, 6 (Mar 1997), 896–902.
[15] D. S. Cao, J. C. Zhao, Y. N. Yang, C. X. Zhao, J. Yan, S. Liu, Q. N. Hu, Q. S. Xu, and Z. Y. Liang. 2012. In silico toxicity prediction by support vector machine and SMILES representation-based string kernel. SAR QSAR Environ Res 23, 1-2 (2012), 141–153.
[16] C. Collaboration. 2010. CMS data processing workflows during an extended cosmic ray run. Journal of Instrumentation 5, 03 (2010), T03006.
[17] 1000 Genomes Project Consortium and others. 2015. A global reference for human genetic variation. Nature 526, 7571 (2015), 68–74.
[18] Corina Cortes and Vladimir Vapnik. 1995. Support-vector machines. Machine Learning 20, 3 (01 Sep 1995), 273–297.
[19] Daniele D’Agostino, Luca Roverelli, Gabriele Zereik, Andrea De Luca, Ruben Salvaterra, Andrea Belfiore, Gianni Lisini, Giovanni Novara, and Andrea Tiengo. 2017. Nextflow enables reproducible computing on Large Clusters. Commun. ACM 51, 1 (Jan 2018), 107–113.
[20] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo P. Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. Nature Biotechnology 35, 4 (11 April 2017), 316–319.
[21] Payam Emami Khoonsari, Pablo Moreno, Sven Bergmann, Joachim Burman, Marco Capuccini, Matteo Carone, Marta Cascante, Pedro de Atauri, Carles Foguet, Alejandra Gonzalez-Beltran, Thomas HankeReiner, Kenneth Haug, Sijin He, Stephanie Herman, David Johnson, Namrata Kale, Anders Larsson, Steffen Neumann, Krzysztof Peters, Luca Pireddu, Philippe Rocca-Serra, Pierrick Roger,
[23] A. Khan. 2017. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Computing*, 4, 5 (September 2017), 42–48.

[24] Grzegorz Kudla, Leszek Lipinski, Fanny Caffin, Aleksandra Helwak, and Maciej Zylcz. 2006. High Guanine and Cytosine Content Increases mRNA Levels in Mammalian Cells. *PLoS Biology* 4, 6 (05 2006).

[25] Kyung-Ha Lee, Yoon-Joon Lee, Hyunsuk Choi, Youn Dohn Chung, and Bongki Moon. 2012. Parallel Data Processing with MapReduce: A Survey. *SIGMOD Rec.* 40, 4 (Jan. 2012), 11–20.

[26] Mark McGann. 2011. FRED Pose Prediction and Virtual Screening Accuracy. *Journal of Chemical Information and Modeling* 51, 3 (2011), 578–596. PMID: 21233318.

[27] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Res.* 20, 9 (Sep 2010), 1297–1303.

[28] Jill P. Mesirov. 2010. Accessible Reproducible Research. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST ’10). IEEE Computer Society, Washington, DC, USA, 1–10.

[29] Ulf Norinder and Scott Boyer. 2016. Conformal Prediction Classification of a Large Data Set of Environmental Chemicals from ToxCast and Tox21 Estrogen Receptor Assays. *Chemical Research in Toxicology* 29, 6 (2016), 1003–1010.

[30] Ulf Norinder and Scott Boyer. 2017. Binary classification of imbalanced datasets using conformal prediction. *Journal of molecular graphics and modelling* ?2 (March 2017), 256–265.

[31] Martin Odersky, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Steimann, Matthias Zenger, and et al. 2004. An overview of the Scala programming language. Technical Report.

[32] Open Container Initiative. 2016. The 5 principles of Standard Containers. https://github.com/opencontainers/runtime-spec/blob/master/principles.md. (Dec. 2016). Accessed 06-04-2018.

[33] D. Seeliger and B. L. de Groot. 2010. Ligand docking and binding site analysis with PyMOL and Autodock/Vina. *J. Comput. Aided Mol. Des.* 24, 5 (May 2010), 417–422.

[34] Alan Shimel. 2016. Docker becomes de facto Linux standard. http://www.networkworld.com/article/2226751/opensource-subnet/docker-becomes-de-facto-linux-standard.html. (April 2016). Accessed 05-12-2017.

[35] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST ’10)*. IEEE Computer Society, Washington, DC, USA, 1–10.

[36] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 European UNIX Users’ Group Conference*. 241–248.

[37] Teague Sterling and John J. Irwin. 2015. ZINC 15 – Ligand Discovery for Everyone. *Journal of Chemical Information and Modeling* 55, 11 (2015), 2324–2337. PMID: 26479676.

[38] Fredrik Svensson, Ulf Norinder, and Andreas Bender. 2017. Modelling compound cytotoxicity using conformal prediction and PubChem HTS data. *Toxicol. Res.* 6 (2017), 73–80. Issue 1.

[39] S. Tore, M. Lindberg, I. Falman, A. Vallin, O. Mohill, P. Freyhult, L. Nilsson, M. Agback, L. Viklund, H. Zazzik, O. Sjöuth, M. Capuccini, J. Mååller, D. Murtagh, and A. Hellander. 2017. SNIC Science Cloud (SSC): A National-Scale Cloud Infrastructure for Swedish Academia. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. 219–227.

[40] S. van der Walt, S. C. Colbert, and G. Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering* 13, 2 (March 2011), 22–30.

[41] Chen Wu, Rodrigo Tobar, Kevin Vinsen, Andreas Wicenec, Dave Pallot, Baoqiang Lao, Ruonan Wang, Tao An, Mark Boulton, Ian Cooper, Richard Dodson, Markus Dolenksy, Ying Mei, and Feng Wang. 2017. DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge. *CoRR* abs/1702.07617 (2017). arXiv:1702.07617

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Re- sident Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI’12)*. USENIX Association, Berkeley, CA, USA, 2–2.

[43] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59 (Oct. 2016), 56–65.

[44] D. Zhao, M. Mohamed, and H. Ludwig. 2018. Locality-aware Scheduling for Containers in Cloud Computing. *IEEE Transactions on Cloud Computing* PP, 99 (2018), 1–1.