MIPRO BEST PAPER AWARD

Cloudflow – A Framework for MapReduce Pipeline Development in Biomedical Research

Lukas Forer[1,2], Enis Afgan[3,4], Hansi Weißensteiner[1,2], Davor Davidović[3], Günther Specht[2], Florian Kronenberg[1], Sebastian Schönherr[1,2],
[1]Division of Genetic Epidemiology, Medical University of Innsbruck, Innsbruck, Austria
[2]Institute of Computer Science, Research Group Databases and Information Systems, Innsbruck, Austria
[3]Center for Informatics and Computing, Ruđer Bošković Institute, Zagreb, Croatia
[4]Department of Biology, Johns Hopkins University, Baltimore MD, USA
sebastian.schoenherr@i-med.ac.at

Abstract - The data-driven parallelization framework Hadoop MapReduce allows analysing large data sets in a scalable way. Since the development of MapReduce programs can be a time-intensive and challenging task, the application and usage of Hadoop in Biomedical Research is still limited. Here we present Cloudflow, a high-level framework to hide the implementation details of Hadoop and to provide a set of building blocks to create biomedical pipelines in a more intuitive way. We demonstrate the benefit of Cloudflow on three different genetic use cases. It will be shown how the framework can be combined with the Hadoop workflow system Cloudgene and the cloud orchestration platform CloudMan to provide Hadoop pipelines as a service to everyone.

The framework is open source and free available at https://github.com/genepi/cloudflow.

I. INTRODUCTION

Since the advent of high-throughput technologies in the field of molecular biology (i.e. Next Generation Sequencing (NGS)), even more data is produced and needs to be analysed. Thus, molecular biology has envolved into a big data science, where the bottleneck is no longer the production of raw data in the laboratory, but its analysis and interpretation [1]. To scale with the increasing data volume and the number of available resources, workflows need to be parallelized efficiently. MapReduce and Cloud Computing constitute an attractive alternative to deal with the large datasets [2]. However, writing a MapReduce job can be a challenging task that prevents domain experts from using such models in their daily work. Additionally, the reusability of the mapper and reducer functions is limited, resulting in a use-case specific implementation for every problem.

Existing high-level languages on top of Hadoop facilitate the development process and allow writing MapReduce jobs in form of queries. For example, Apache Pig (http://pig.apache.org/) provides a compiler to translate such queries into an execution plan, which is then automatically translated into a sequence of MapReduce jobs. Apache Pig provides interfaces for *filter*, *group* and *join* operations, extendible for application-specific logic using user defined functions (UDFs). However, Apache Pig has been developed to analyze datasets based on the relational model. The execution of complex calculations across several rows is limited. This implies the consequence that users have to write complex UDFs. Again, such functions need to be implemented by the end users and can be, similar to native MapReduce jobs, hard to implement, test and maintain. Several projects (e.g. SeqPig [3] or BioPig [4]) make use of UDFs and provide a collection of Pig scripts to researchers in Genetics. The goal is to provide ready-to-use workflows to end users, which can be then adapted to their use-case. Nevertheless, combining different scripts to a pipeline or reusing existing blocks does not build a key feature.

To overcome this issue, FlumeJava [5] proposed a new concept to compose pipelines based on immutable parallel collections where several operations can be used to process them in a parallel way. This concept was successfully implemented in Apache Crunch (https://crunch.apache.org/), which executes the pipelines as Hadoop MapReduce jobs. However, the utilization of such pipelines in Bioinformatics is still limited.

In this paper we present Cloudflow, a MapReduce pipeline framework, which is based on a similar concept as proposed by [5]. In contrast to existing approaches, Cloudflow was developed to simplify the pipeline creation in biomedical research, especially in the field of genetics. For that purpose Cloudflow supports a variety of NGS data formats and contains a rich collection of built-in operations for analyzing such kind of datasets (e.g. quality checks, mapping reads or variation calling). The main concept behind our approach is to break complex data analysis steps into three basic operations. All further use-case specific operations are built by implementing or extending one of the basic operations. Pipelines are then composed by creating a sequence of these operations. The framework itself translates the set of pipeline operations into one or more MapReduce jobs and decides which of the operations are executed in the *map* or in the *reduce* phase. Thus, Cloudflow hides the complexity and the implementation details of MapReduce jobs allowing scientists to build pipelines via an intuitive method. Moreover, Cloudflow can be utilised in combination with the workflow system Cloudgene [6]. To validate our approach, we developed three Genetics data-analysis pipelines with Cloudflow. The results demonstrate that our contribution (a) helps minimizing the development time, (b) increases the reusability of code, and (c) creates only a minimal overhead in terms of execution time compared to an identical MapReduce implementation.

II. MapReduce Background

MapReduce is a parallel programming model introduced by Google in 2004 with the aim to develop a simple and scalable method to process large datasets on several machines in parallel. The main idea behind this distributed programming model is that a long-time calculation is split into a map and a reduce phase which contains all the logic behind the calculation and is specified by the user. The underlying framework itself takes care of parallelization, task scheduling, load-balancing and fault-tolerance. Due to its simplicity, MapReduce is used to solve many scientific problems where large-scale computing is needed. Moreover, MapReduce is ideal for parallel batch processing of terabytes of input data. The data-flow of a MapReduce program consists of several steps, where only the map and reduce function are problem-specific and the other steps are loosely coupled with the problem and generalized. In the first step, the input data set is split into key/value pairs and a user defined map function is executed for each pair:

$$map(key, value) \rightarrow list((key_i, value_i)) \text{ , where } i = 0 \ldots n$$

The map function reads a pair, performs some problem-specific calculations and produces zero or *n* intermediate key/value pairs for each input pair. In the next step, the intermediate key/value pairs are grouped by similar keys and a merged list of all values for this key is created. Finally, the user-defined reduce function is applied to the intermediate key/list pairs:

$$reduce(key_i, list(value_i)) \rightarrow list(outvalue)$$

The values created by the reduce function are the final outputs of a MapReduce job.

A. Apache Hadoop

Apache Hadoop is an open source project including several sub-projects for distributed computing. It includes the most widely used open-source implementation of Google's MapReduce framework (simply called MapReduce), the distributed file system (HDFS), and several other sub-projects. In general, all previous discussed features of the original MapReduce approach are implemented in Apache Hadoop. Writing a MapReduce job with Hadoop can either be done directly in Java by implementing the relevant methods or by using Hadoop's streaming mode. This allows specifying a script or any executable as the mapper or reducer. Two different versions of MapReduce are available. MRv1 includes a namenode (centerpiece of HDFS), a secondary namenode (merging logs into a file system snapshot), a job-tracker (job assignment) and several task-trackers (workers). MRv2 or Hadoop YARN, splits the work of the job-tracker (resource management, job scheduling) into two different daemons, namely a resource manager and an application master. Additionally a node manager is introduced to manage the user processes per node. The application master describes a framework that works together with the node manager and the resource manager to monitor and to execute tasks. The idea behind this new model is that applications using frameworks different than MapReduce (e.g. Apache Spark) can be executed via YARN as well.

B. High-Level Languages based on MapReduce

To simplify the implementation of MapReduce jobs, Apache Pig has been introduced. It is based on HDFS and MapReduce and allows a fast implementation of data flows with already available data operations such as *join*, *filter* or *group by*. Data flows are specified in the Pig Latin language that describes a directed acyclic graph (DAG) and defines how data should be processed. A further advantage of Apache Pig is the ability to check the data flow on optimizations, e.g. if two grouping statements can be combined. The costs to write code in Apache Pig are lower than setting up a Java project and implementing the functions. But as stated earlier, Apache Pig includes only a limited number of operators and writing Apache MapReduce jobs directly in Java yields to advantages in speed compared to Apache Pig.

C. MapReduce Pipelining

The pipeline framework FlumeJava [5] is based on the concept of immutable parallel collections. This kind of data-structure can be used to process and analyze their items in parallel. The end user can either use one of the predefined functions or can combine them with its own. Each function is implemented as parallel for-each loop, which is than translated by the framework into a series of MapReduce jobs. During the translation, the framework itself decides if such a function should be executed locally (i.e. sequentially) or remotely (i.e. parallel). Apache Crunch is a freely available open source implementation of FlumeJava.

III. Cloudflow

The overall idea behind Cloudflow is to simplify the creation of analysis pipelines by encapsulating complex data analysis steps in simple operations. This approach helps hide the complexity and the implementation details of complex data-parallel pipelines. Moreover, the concept of using basic operations increases the reusability and enables the testing of the operation logic on a local workstation by using existing unit testing frameworks.

Since Cloudflow uses the Hadoop framework for pipeline execution, it offers parallel data processing, data reliability and fault tolerance out of the box. This fact is especially important in the field of Cloud Computing, where infrastructure often relies on commodity hardware and nodes can fail on a regular basis (e.g. due to miss-configuration or hardware failures). At the same time, the architecture of Cloudflow is independent from MapReduce; it provides parallelization constructs and abstraction interfaces that can be used to extend the system by implementing other parallel programming models in the future (e.g. translating the operations into Apache Spark jobs).

Instead of developing a new declarative language for the pipeline composition, we developd a clear Java API. The proposed framework implements different patterns to speed up the pipeline creation, to be extensible and to support test-driven pipeline development. The following

section gives an overview on the abstraction, explains the basic operations in detail, and shows how pipelines are created.

A. Data Types and Basic Operations

Cloudflow operates on records consisting of a key/value pair, whereby different record types are available (e.g. TextRecord, IntegerRecord, FastqRecord). A *loader* class is responsible to load the input data and to convert it into an appropriate record type. As mentioned earlier, Cloudflow supports three different basic operations. These are used to analyze and transform records; specifically, *transform*, *summarize*, and *group* operations exist.

The *transform*-operation is used to analyze one input record and to create between *0* and *n* output records. The user implements the computational logic for this operation by extending an abstract class. This class provides a simple function, which is executed by the Cloudflow framework for all input records in parallel:

```
Class MyTransformer extends Transformer {
    public void transform(Record) {
       doSomethingInParallel();
       emit(new Record());
    }
}
```

The *summarizer* operates on a list of records, whereby records with the similar key are grouped. Thus, the signature of the process method has the key and a list of records as an input:

```
Class MySummarizer extends Summarizer {
   public void summarize(Key, List<Record>) {
       doSomethingInParallel();
       emit(new Record());
    }
}
```

The *group*-operation is a special operation, which takes a list of records as an input and creates a *Record* group with the same key. Our framework inserts automatically a group-operation between a transform- and a summarize-operation. This ensures, that output records of the *transform* operation are compatible with the input records of the *summarize* operation. The *group*-operation is realized by using the shuffle phase of a MapReduce job.

Based on these three operations, the user defines pipelines by building sequences of operations. A pipeline has to start with a transform-operation, all further operations are optional and can be used in arbitrary order.

B. Extended Operations

Complex operations are built by combing one or more basic operations. We already implemented several standard operations that are helpful for the analysis of text or numerical data records:

- *Filter*: this operation is a special *transform*-operation, which emits the record to the subsequent operation iff a user-defined condition is fulfilled.
- *Split*: the *transform*-operation calculates for each input record a new split level (i.e. a new key). This key is used by the group-by operation to create chunks, which can then be analyzed by a user-defined summarize-operation.
- Aggregation (sum, mean): This defines a *group-by*-operation followed by a *summarize*-operation to aggregate all values with the same key (e.g. calculates the mean of all values). One record with the aggregated value is then emitted to the subsequent operation.
- *Executor:* this *summarize*-operation writes all grouped values into a file on the local disk. It is then used as the input of an external UNIX command line program. Based on the lines of the output file, new records are created and emitted to the subsequent operation.

Since Cloudflow's operations are based on the Composite pattern, all these extended operations can also be used as a basis for new operations. In addition, this enables to split complex operations into several sub-operations, which improves testing and maintenance.

C. Pipeline Composition

The user builds pipelines by connecting several operations with compatible interfaces. For this purpose our framework implements the Builder pattern, which enables (a) building complex pipelines, (b) providing type safety and (c) the implementation of domains specific builders (see Section III.D). In addition, the Builder pattern ensures that only a valid sequence of operations can be created (i.e. after the group-by operation a summarize operation has to be added).

```
Class LineToWords extends Transformer {
   public void transform(TextRecord rec) {
    String[] words = rec.getValue().split()
    for (String word: words){
      emit(new IntegerRecord(word, 1));
    }
   }
}

pipeline.loadText(input)
        .transform(LineToWords.class)
        .sum()
        .save(output);
```

Listing 1. WordCount Example using Cloudflow

To help the user and accelerate the pipeline composition process, Cloudflow provides already a set of useful operations. This has the advantage that even a default WordCount example can be broken down into a few simple operations and is defined in a single line of code (see Listing 1). In a first step, the text file is loaded from HDFS (*loadText*). Then, for each record (i.e. line of input) the application-specific *LineToWords* operation is executed, which splits the line into words and creates a new record efor ach word. In the last step a predefined *sum* operation is executed. It extends the pipeline by a *group-by* operation and a *summarize*-operation in order to sum up all the values for a certain key.

For frequently used operations (e.g. sum, mean or count), we created special builder functions, which extend the pipeline and improve the code readability by keeping the code simple.

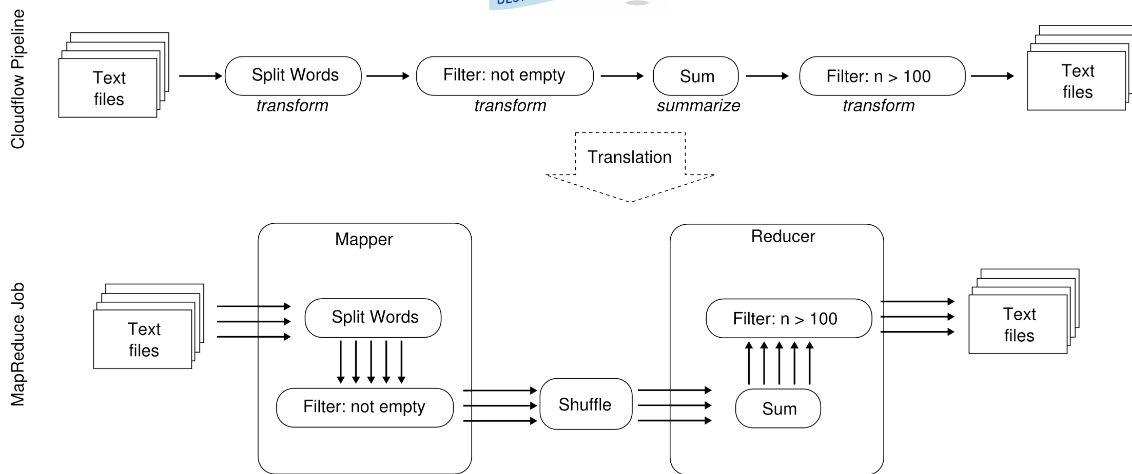

Figure 1. Cloudflow translates the operation sequence automatically into an executable MapReduce job

D. Pipeline Execution

Before the execution, Cloudflow checks the compatibility of input and output records of consecutive operations. This ensures that only valid and executable pipelines are submitted to a Hadoop cluster.

If the pipeline is executable and valid, then the operation sequence is translated into an execution plan, that decides if an operation is executed in the *map* or in the *reduce* phase. Based on this plan, Cloudflow creates one or more MapReduce jobs and configures them to execute the user-defined operations in the correct order. In this translation step, Cloudflow tries to minimize the number of MapReduce jobs by combining consecutive *transform*-operations and by executing all *transform*-operations after a *summarize*-operation in the same reducer instance (see Figure 1).

For additive *summarize*-operations (e.g. sum), Cloudflow takes advantage of Hadoop's combiner functionality. The idea of this improvement is to combine the key/value pairs that are generated by all map tasks on the same machine, into fewer pairs. Thus, the number of pairs that are transferred between mapper and reducer are minimized, which results in a positive effect on the network bandwidth since useless communication is avoided.

IV. CLOUDFLOW FOR BIOINFORMATICS

Cloudflow provides a variety of already implemented utilities, which facilitate the creation of pipelines in the field of Bioinformatics (especially for NGS data in Genetics). For that purpose, we implemented, based on HadoopBAM [7], several record types and loader classes in order to process FASTQ, BAM and VCF files. Moreover, we created several operations and filters for the analysis of biological datasets (see Table I for an overview of all currently implemented operations and filters).

For example, a typical quality control pipeline for VCF files can be implemented by simple combining of several built-in operations. First, we apply predefined filters to discard variations that are monomorphic, marked as duplicates, or are Insertions or Deletions (InDels). For all records passing the filters, Cloudflow applies a *summarize*-operation that calculates the call rate for each variation (see Listing 2).

```
Class CallRateCalc extends Transformer {
  public void transform(VcfRecord record) {
    VariantContext snp = record.getValue();
    float call = callRate(snp);
    emit(new FloatRecord(snp.getID(), call);
  }
}

pipeline.loadVCF(input)
        .filter(MonomorphicFilter.class)
        .filter(DuplicateFilter.class)
        .filter(InDelFilter.class)
        .transform(CallRateCalc.class)
        .save(output);
```

Listing 2. VCF Quality Control Pipeline using Cloudflow

V. DEPLOYING PIPELINES AS A SERVICE

Cloudgene [6] is a web-based platform to create and execute workflows consisting of Hadoop MapReduce, Apache Pig and command line-based programs. It can be seen as an additional layer between Hadoop MapReduce and the end user that hides the complexity of the MapReduce framework. Therefore, Cloudgene is the perfect candidate to provide Cloudflow pipelines as a service. Such pipeline can be integrated into the workflow platform by utilizing Cloudgene's plugin interface. No adaptation to the source code is needed, while only a simple plain text file including a header, input parameters, output parameters and the definition of the workflow itself need to be created. When launching Cloudgene, the manifest file is loaded and the client interface is automatically rendered using information from the file. As Cloudgene supports different technologies, it is possible to parallelize the calculations using Cloudflow and to visualize the results using R.

Cloudflow requires a compatible MapReduce cluster for executing pipelines. CloudMan [8] makes it possible to easily procure and configure a functional data analysis platform on a cloud infrastructure. The procured platform delivers a scalable cluster-in-the-cloud and a data analysis environment preconfigured with a number of applications. With its ability to be launched and managed via a web browser on a number of clouds, customized as necessary, and easily shared with collaborators, CloudMan makes it

TABLE I. CURRENTLY SUPPORTED DATA FORMATS AND OPERATIONS

Data Format		Pipeline Operation	Description
Fastq	Split	`split()`	Find pairs (for paired-end reads)
	Filter	`filter(LowQualityReads.class)`	Filters reads by quality
		`filter(SequenceLength.class)`	Filters reads by sequence length
	Other	`findPairedReads()`	Detects read pairs
		`align(referenceSequence)`	Aligns sequences against a reference (using jBWA for alignment)
BAM	Split	`split()`	Creates fixed size chunks (e.g. 64 MB)
		`split(5, BamChunk.MBASES)`	Creates logical chunks (e.g. 5MBases)
	Filter	`filter(UnmappedReads.class)`	Filters unmapped reads
		`filter(LowQualityReads.class)`	Filters reads by map.quality
	Other	`findVariations()`	Finds variations in aligned reads (using samtools)
VCF	Split	`split()`	Creates fixed size chunks (e.g. 64 MB)
		`split(5, VcfChunk.MBASES)`	Creates logical chunks (e.g. 5MBases)
	Filter	`filter(MonomorphicFilter.class)`	Filters monomorphic site
		`filter(DuplicateFilter.class)`	Filters duplicates
		`filter(InDelFilter.class)`	Filters inDels
		`filter(CallRateFilter.class)`	Filters by call rate
		`filter(MafFilter.class)`	Filters by MAF
	Other	`checkAlleleFreq(reference)`	Allele frequency check with external reference (e.g. 1000 genomes)

possible to readily utilize cloud resources in a research environment. The approach on how Cloudgene and CloudMan can be combined efficiently have already been demonstrated [9].

VI. EVALUATION

To evaluate our approach, we implemented three different Bioinformatics data-analysis pipelines using Cloudflow and integrated them into Cloudgene. The results of our experiments demonstrate that Cloudflow has only a minimal overhead in the execution time compared to an identical pure MapReduce implementation. However, the performance of Cloudflow is better than Apache Crunch's (see Figure 2).

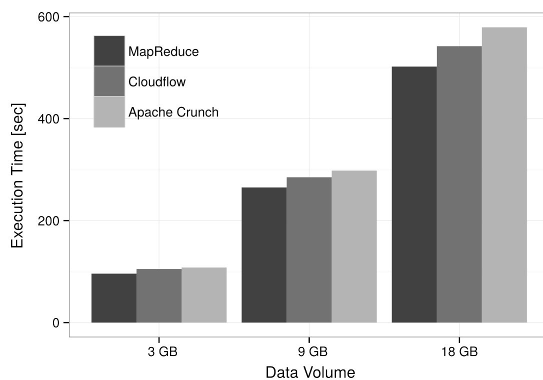

Figure 2. Execution Time of a Cloudflow pipeline, an Apache Crunch pipeline and a pure MapReduce implementation of WordCount

A. *Preprocessing and Mapping*

When working with NGS data, the quality of raw data needs to be checked before a successful subsequent downstream analysis (e.g. read mapping/alignment) can be achieved. The overall goal of alignment is to dock the vast amount of short reads, mostly in the FASTQ file format, to a reference genome. Factors such as read errors, insertions or deletions of bases must be considered that finally results in the most accurate genome position for each read.

The goal of the following pipeline is the alignment of paired-end data to a reference genome. Therefore, the FASTQ data is loaded using the FastqLoader, which creates records for each sequence. The records are then filtered by an average base quality of 30. Numerous other quality metrics (such as sequence length or C/G content) can be filtered as well. Since paired-end reads are used, read pairs are detected using a predefined *transform*-operation. The aligner step is implemented as a *summarize*-operation that calls a parallelized version of BWA-MEM [10]. This has been achieved by using JNIs (https://github.com/lindenb/jbwa). Similar to BWA-MEM 99,100 reads are aligned to the user-specified reference genome in one batch. Aligned reads are saved in HDFS in the BAM file format (see Listing 3.A).

B. *Variation Calling*

After data has been aligned and cleaned (e.g. removing duplicates, quality recalibration), the next step of NGS pipelines is the detection of reliable variants that can be used e.g. in association studies. A widely used pipeline for variant detection is GotCloud, developed at the Center of Statistical Genetics (University of Michigan) and utilized in the 1000 Genomes (1000G) project.

In this example pipeline, the aim is to find variations without a statistical model by implementing a simple counting approach of the four bases. This is only possible when using high coverage data. Therefore, in the first step the BAM file (created in the previous pipeline) is loaded and chunked in user-specified splits (e.g. 5 MBases). Variations are then detected for each chunk by counting

the occurrences of A, C, G, T on each position. Finally, the detected variations are stored in VCF files (see Listing 3.B).

C. Genome-Wide Association Studies

Many genome-wide association studies (GWAS) have identified associations between various phenotypes and common sequence polymorphisms, which might play a role for disease development. Technologies like microarrays have made it possible to measure millions of single nucleotide polymorphisms (SNPs) of one individual simultaneously and for low cost. Since the costs of microarrays is much lower than next-generation sequencing (NGS), it is today the cheapest method to genotype large-scale population studies. Such datasets are combined with collected phenotypes (e.g. diseases and measured data) in order to detect if one of these variations has a high impact on the value of a phenotype. Since the size of such datasets grows rapidly, a parallelization at the data-level is necessary to analyze the data in appropriate time.

```
//A. Preprocessing and Mapping
pipeline.loadFastq(input)
        .filter(LowQualityReads.class, 30)
        .findPairedReads()
        .align(refSeq)
        .save(output);

//B. Variation Calling
pipeline.loadBam(input)
        .split(5, ChunkSize.MBASES)
        .groupByKey()
        .findVariations(refSeq)
        .save(output);

//C. Genome-Wide Association Study
pipeline.loadText(input)
        .split(1000, ChunkSize.LINES)
        .execute(SnpTestExecutor.class).
        .filter(FilterHeader.class)
        .filter(FilterInvalidSnps.class)
        .save(output);
```

Listing 3. Complete NGS pipeline using Cloudflow

The parallelization of the association analysis was realized by splitting the list of markers into chunks. In detail, the mapper splits all input SNPs into chunks with a fixed number of SNPs (e.g. 1000). Then, the reducer executes the linear regression model for each chunk by using SNPTest. Finally, the reducer collects the results and merges them into a single file. The corresponding Cloudflow pipeline loads the text input file and automatically creates records for each line. On these records we apply the *split* operation, which creates chunks containing a fixed number of lines. For the execution of the SNPTest program, we can implement a special operation called *BinaryExecutor,* which enables us to write the chunks automatically to the POSIX file system. In the next step, we can use this file as the input file for SNPTest. After the execution the operation creates text records for each line of results (see Listing 3.C).

VII. Conclusion

Cloudflow's overall aim is to simplify the development of complex MapReduce pipelines by abstracting the map and the reduce function from end users. Therefore, operations need only be written once and can be re-used for future MapReduce usage. The major advantage of Cloudflow lies in the provision of validated operations, especially in the area of genetics, and its extensibility. Combining Cloudflow with CloudMan (cluster orchestration) and Cloudgene (Hadoop workflow system) allows users to use Hadoop without a deeper knowledge of the internal MapReduce concepts and could yield to a boost of Hadoop in genetics.

Acknowledgment

This work was, in part, supported by the "Scalable Big Data Bioinformatics Analysis in the Cloud" grant from the Croatian Ministry of Science, Education, and Sport and the Austrian Federal Ministry of Science and Research (BMWF) and by the FP7-PEOPLE programme grant 277144 (AIS-DC).

References

[1] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, pp. 255–260, 2013.

[2] S. Yazar, G. E. C. Gooden, D. A. Mackey, and A. W. Hewitt, "Benchmarking undedicated cloud computing providers for analysis of genomic datasets.," *PLoS One*, vol. 9, no. 9, p. e108490, Jan. 2014.

[3] A. Schumacher, L. Pireddu, M. Niemenmaa, A. Kallio, E. Korpelainen, G. Zanetti, and K. Heljanko, "SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop.," *Bioinformatics*, vol. 30, no. 1, pp. 119–20, Jan. 2014.

[4] H. Nordberg, K. Bhatia, K. Wang, and Z. Wang, "BioPig: a Hadoop-based analytic toolkit for large-scale sequence data.," *Bioinformatics*, p. btt528–, Oct. 2013.

[5] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava," *ACM SIGPLAN Not.*, vol. 45, no. 6, p. 363, May 2010.

[6] S. Schönherr, L. Forer, H. Weißensteiner, F. Kronenberg, G. Specht, and A. Kloss-Brandstätter, "Cloudgene: A graphical execution platform for MapReduce programs on private and public clouds," *BMC Bioinformatics*, vol. 13, no. 1, p. 200, 2012.

[7] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko, "Hadoop-BAM: Directly manipulating next generation sequencing data in the cloud.," *Bioinformatics*, p. bts054–, Feb. 2012.

[8] E. Afgan, B. Chapman, and J. Taylor, "CloudMan as a platform for tool, data, and analysis distribution.," *BMC Bioinformatics*, vol. 13, no. 1, p. 315, Jan. 2012.

[9] L. Forer, T. Lipic, S. Schonherr, H. Weisensteiner, D. Davidovic, F. Kronenberg, and E. Afgan, "Delivering bioinformatics MapReduce applications in the cloud," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*, 2014, pp. 373–377.

[10] H. Li, "Aligning sequence reads , clone sequences and assembly contigs with BWA-MEM," vol. 00, no. 00, pp. 1–3, 2013.