Università degli Studi di Cagliari
Dottorato di Ricerca in Ingegneria Biomedica
Ciclo XXVII

Enabling Data-Intensive Biomedical Studies

Presentata da:	Simone Leo
Coordinatore dottorato:	Prof. Giacomo Cao
Tutor:	Prof. Riccardo Scateni
Relatore:	Dr. Gianluigi Zanetti

Settore scientifico-disciplinare di afferenza: ING-INF/06
Esame finale anno accademico 2013–2014

Abstract

The constantly increasing size and complexity of datasets involved in biomedical projects is deeply transforming approaches to their solution. Large scale studies require specifically designed computational frameworks that are capable of fulfilling many diverse requirements, the most important of which can be summarized in the fundamental properties of scalability, reproducibility and traceability. Although in recent years several new technologies have emerged that help deal with the issues raised by data-intensive research projects, applying them to the construction of a computational solution for the specific problem at hand is far from trivial, as no one-size-fits-all recipe exists for such a task. This work describes a methodology for approaching this new class of studies through several examples of solutions applied to concrete research problems.

Contents

Chapter 1

Introduction

In the recent past, the exponential growth of data produced in numerous industries and scientific disciplines has led to a whole new class of challenges related to their storage, processing, sharing, visualization and protection. Buzzwords that have attempted to capture this phenomenon include *data deluge*, *data revolution* and — more recently — *big data*.

Despite being quite general and loosely defined, all these terms apply to problems where datasets reach a critical size, above which traditional processing methods such as desktop applications and relational database management systems (RDBMS) cease to be effective. Spurred by the rapid diffusion of data gathering devices, which has been accompanied by a constant decrease in storage media costs, data-intensive applications require specific technologies (most notably distributed computing), infrastructure (data centers, multi-node clusters, networking) and professional background. The exact size that makes data “big” is, of course, highly dependent on the specific field and organization, and varies with time.

Although the data revolution is often associated with large web services such as social networks and e-commerce platforms, a consistent part of the transformation concerns science and its methods. Indeed, data-intensive scientific discovery is regarded by some authors as a “fourth paradigm” that follows empirical, theoretical and computational science [25]. This scenario is marked not only by the introduction of radically new computational tools that are capable of dealing with the immense amounts of data being generated, but also by an equally fundamental change in the way scientific research and communication is conducted [64]. Scientific papers and technical reports typically present their results in a highly condensed form, often just the tip of the iceberg of the vast amount of data involved in the analysis process. This intermediate data is usually discarded after publishing the manuscript or when the supporting grant ends, together with the accompanying metadata

— experimental set-up, software version and parameters, data provenance and interdependence — provided it has been recorded at all. This loss of information can be due to several reasons, including insufficient storage space, lack of community standards, inadequate data management tools or even natural disasters [36]. A recent survey [62] of data sharing practices among scientists indicates insufficient time and funding as the main reasons for not making data electronically available, while over half of the respondents reported that they did not use any metadata standards. Whatever the cause for this failure to properly preserve and share experimental data, the end result is a missed opportunity for reuse, which can easily lead to new findings comparable to — or more significant than — the original ones [37, 64].

As research projects become more data driven, their requirements in terms of computational literacy and infrastructure progressively increase, to the point that it's hardly possible to do good research without good software [21]. In some cases, low quality software can even lead to disastrous errors, capable of fundamentally compromising the main results [41]. In data-intensive research, however, ensuring correctness is only part of the problem: issues like efficiency, scalability and reproducibility are of equal importance, and must be tackled with specifically designed computational solutions.

1.1 Big Data in the Life Sciences

While some scientific fields, such as high-energy physics and astronomy, have been dealing with large collections of data for a long period, life sciences are relatively new to the game. Since the advent of high-throughput genomics, the amount of data being managed by major bioinformatics institutions has been growing exponentially, and is currently in the order of the petabytes [37]. In little more than a decade, next-generation sequencing (NGS) platforms have deeply transformed genome analysis and its applications to variation detection, RNA sequencing (RNA-seq), whole genome genotyping, *de novo* assembly and more, greatly increasing data generation throughput — the current order of magnitude is of several gigabases (Gb) per day — while lowering costs [48, 57]. High-density genotyping microarrays [29], with their ability to test for millions of genetic variants in parallel, are yet another example of devices capable of turning labs into big data producers, especially when used for genome-wide association studies (GWAS) [13], where large sample sizes are required for statistical power.

An equally disruptive transformation is underway in health care with the ongoing development of what has been called P4 medicine (predictive, preventive, personalized and participatory) [20]. The overall goal is to integrate data from multiple sources such as whole genome sequencing, electronic health records (EHR) and wearable sensors to build predictive models usable for prevention, so that disease-inducing patterns may be detected via machine learning and corrected before symptoms appear. Due to the high incidence of health-associated costs in major economies, systematic use of big data analytics in medicine is expected to generate huge savings [43].

Genomics and medicine, however, are not the only life sciences where researchers have to deal with large amounts of data. Neuroscience has joined the revolution as well, bringing in specific issues of extreme heterogeneity, multiple organisms, high dimensionality and lack of horizontal integration [56]. In botany, scientists are rapidly moving towards multi-omics approaches as the number of new plant genomes sequenced per year is getting close to one hundred [11], thus intensifying the need for standardization techniques that support automated aggregation and retrieval [66]. In environmental science, efforts aimed at systematically managing heterogeneous, multi-scale and geographically distributed information have led to the development of DataONE [40], a cyberinfrastructure that offers services such as data preservation, replication, access and visualization.

Data-driven research projects stemming from the above scenarios present peculiar challenges that must be met with specifically designed technology. In life sciences, however, data analysis is often still performed with the same simple, traditional instruments that were adequate up to a few years ago, such as quick-and-dirty R scripts for biostatistics processing, spreadsheets for collecting and organizing metadata, or even handwritten notes with no digital counterpart. This started to change in the recent past due to the introduction of several technologies that, being capable of addressing one or more big-data-related issues, can be used as the building blocks of computational frameworks capable of driving the latest generation of large scale research projects.

Galaxy [22], a web platform for biomedical data analysis, provides a standard mechanism to wrap computational tools and datasets in a graphical user interface (GUI), allowing to keep track of parameter choices and execution history. The software has been specifically designed to ensure accessibility and reproducibility, and to facilitate the sharing of analysis workflows and experimental results (see Sec. 1.3).

One problem that Galaxy does not solve, however, is the efficient application of computationally intensive procedures (such as sequence alignment) to large datasets, where the running time can easily become a bottleneck for the whole study. Up to roughly ten years ago, microprocessor speed grew fast enough to keep up with the increasing demands of most applications, but this trend reached saturation as physical limits forced manufacturers to add more CPU cores rather than increase clock frequency [60]. While faster processors provide an automatic performance gain to any (compute-bound) software program, to reap the benefits of multi-core architectures developers must substantially rewrite their applications in order to provide suitable parallel implementations. Moreover, for data-intensive analysis, the number of computing cores available on a single machine may be insufficient. In this case, to bring execution times down to acceptable levels, the workload must be distributed across a network of computing nodes, usually in the form of a Beowulf cluster [59]. Distributed computing, however, presents specific challenges — such as data distribution, load balancing, machine failure handling and inter-machine communication — that have long prevented its adoption by non-experts. The MapReduce[17] programming model and infrastructure, originally designed at Google and popularized by its implementation in the open source Hadoop platform[1], has played a major part in making distributed computing accessible to a wider audience, allowing programmers to concentrate on the core algorithm while the framework takes care of most lower level details (see Sec. 1.2). Due to its cost-effectiveness and suitability for batch processing, Hadoop is widely used in the bioinformatics community, especially for the analysis of NGS data [44, 61, 69].

Although Galaxy provides a standard way to keep track of computational flows, traceability in biomedical applications needs to be managed at all stages, including data acquisition, storage, aggregation, querying and transformation. OME Remote Objects (OMERO) [4], originally developed for microscopy images, allows to manage both data and metadata under a common middleware and application program interface (API), providing access control, annotation, indexing, rendering and processing regardless of file format and programming environment. OMERO.biobank[2] is a traceability framework for data-intensive life sciences built upon OMERO's core services, currently under active development at CRS4[3]. OMERO.biobank allows to manage biological samples, instrumental acquisitions and analysis results in the context of large scale experiments, modeling data generation processes as chains of entities and transforming actions. OMERO and OMERO.biobank are described in more detail in Sec. 1.4.

[1]http://hadoop.apache.org

[2]https://github.com/crs4/omero.biobank

[3]http://www.crs4.it

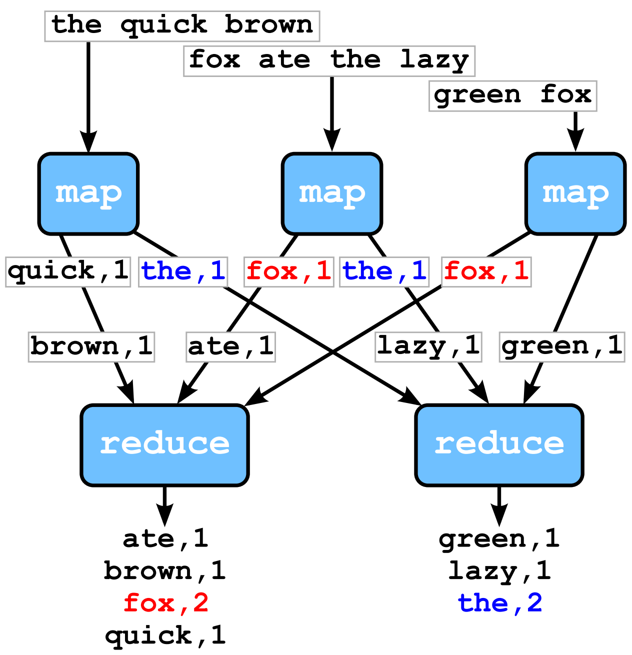


Figure 1.1 – Schematic diagram of a MapReduce application that counts the occurrence of each word in a text dataset. The framework groups and sorts intermediate pairs by key, and sends pairs with the same key to the same reducer.

1.2 MapReduce and Hadoop

MapReduce is a programming paradigm and execution model for large scale, distributed data analysis. Initially developed and internally used by Google for web indexing and monitoring applications, the framework started to gain popularity after its initial publication [16] and subsequent open source implementation in what would later become Hadoop [68].

Designed for data-driven applications — where algorithms are often relatively simple, but the overall task is complicated by the size of the input dataset — MapReduce provides an abstraction layer that hides parallelization-related details such as workload distribution and interprocess communication (IPC) in a library, allowing the programmer to express the computation as a pair of functions: *map* and *reduce*. The map function receives an *input* key/value pair and must emit an *intermediate* key/value pair back to the framework; the latter groups and sorts intermediate pairs by key (this is called the *shuffle and sort* phase in Hadoop), sending pairs with the same key to the same reduce task; the reduce function combines the set of values corresponding to each intermediate key and emits an *output* key/value pair.

Algorithm 1 MapReduce word count

```
function MAP($key, value$)
    for $word \leftarrow value$ do
        emit($word$, 1)
function REDUCE($key, values$)
    $count \leftarrow 0$
    for $v \leftarrow values$ do
        $count \leftarrow count + v$
    emit($key, count$)
```

Fig. 1.1 schematically shows the key/value pair flow in the original MapReduce example [16], a typical data-driven application where the computation — counting the occurrence of each word in a given amount of text — is straightforward, but executing it efficiently on a very large dataset (e.g., in the order of the petabytes) is far from trivial. The MapReduce framework breaks down the input data into subsets (called *input splits* in Hadoop) and feeds each one to a separate map task, repeatedly calling the user's map function (see Alg. 1) with a line of text as the input value (the input key is not used in this case). The map function splits each input line into words, then emits the word itself and the number 1 as the intermediate key/value pair. The framework collects all pairs emitted by the map tasks, performs the aforementioned grouping and sorting, and passes them to the reduce tasks. The user's reduce function receives an intermediate key and an iterator over all corresponding values (all equal to 1 in this case), which are summed to get the occurrence. Finally, the framework collects output key/value pairs and writes them out.

In principle, the MapReduce model can be implemented regardless of the file system being used. In practice, to achieve high scalability, the framework is backed up by a high-performance distributed file system: the Google File System (GFS) [19] in the original version and the Hadoop Distributed File System (HDFS) — an open source implementation of GFS — in Hadoop. GFS/HDFS is optimized for reading data in large batches. To reduce the number of client requests and thus network overhead, files are split into very large blocks — four orders of magnitude larger than those of traditional, generic purpose file systems — which are distributed across the whole cluster (with replication, to ensure fault tolerance).

In the following, the DFS acronym will be used to denote a generic GFS-like distributed file system. One of the main reasons for MapReduce's high efficiency is the *data locality* mechanism: by querying the DFS master (called

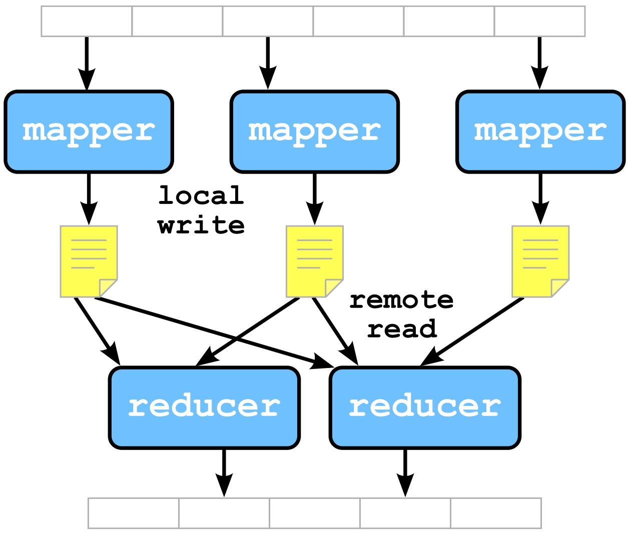
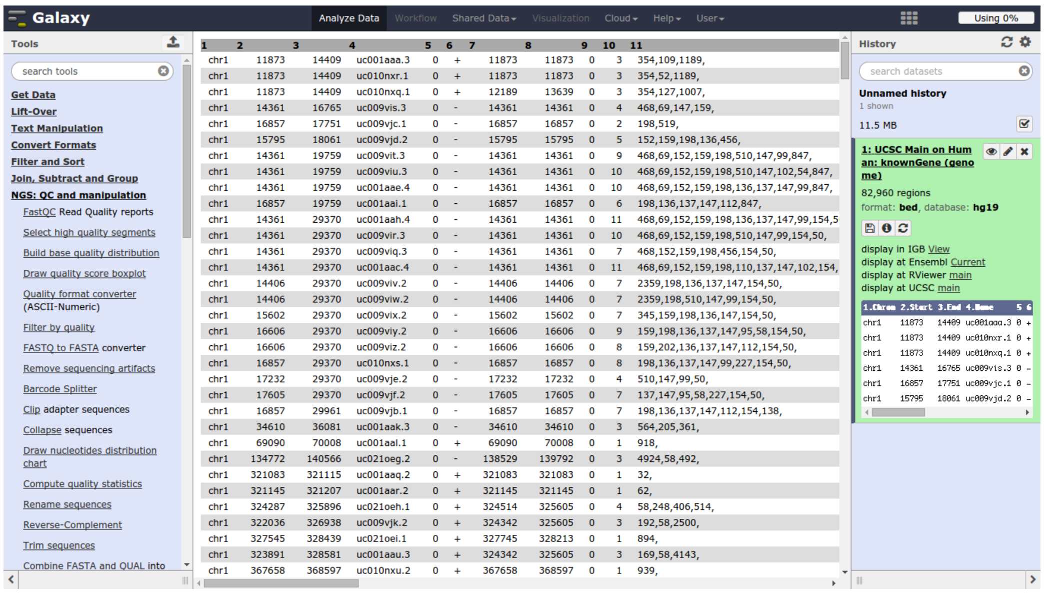


Figure 1.2 – The MapReduce execution model. Input data is read from and output data is written to the distributed file system, while intermediate data is written to the local disks by mappers and remotely read by reducers.

name node in Hadoop) about the locations of the blocks stored in the slaves (*data nodes*), the framework is able to schedule the computation for each split closest to where the corresponding blocks physically reside, thus minimizing data movement across the network. Fig. 1.2 shows a simplified version of the execution model. If not already present, input data is first saved to one or more files in the DFS. When the MapReduce job starts, the framework logically divides input files into splits, schedules an appropriate number of map tasks (usually related to the total number of CPU cores) and assigns a split to each task, which reads from the DFS and writes intermediate data to the local disk. Reduce tasks read intermediate key/value pairs across the network (the set of values corresponding to a given key is scattered among several cluster nodes), apply the reduce function and write the final output back to the DFS. If a task fails, it is automatically rescheduled on a different node, thus providing transparent fault tolerance.

In the ten years after its inception, the Hadoop project has evolved into a whole ecosystem of big data analysis tools that complement and enhance HDFS and MapReduce, including high-level querying and execution engines, distributed databases and many more[4]. Hadoop itself, since version 2, has undergone consistent improvement: HDFS now supports multiple, federated name nodes, and the MapReduce engine has become a specific use case of a more general computing resource manager (called YARN).

[4]`http://hadoopecosystemtable.github.io`



1	2	3	4	5	6	7	8	9	10	11
chr1	11873	14409	uc001aaa.3	0	+	11873	11873	0	3	354,109,1189,
chr1	11873	14409	uc010nxr.1	0	+	11873	11873	0	3	354,52,1189,
chr1	11873	14409	uc010nxq.1	0	+	12189	13639	0	3	354,127,1007,
chr1	14361	16765	uc009vis.3	0	-	14361	14361	0	4	468,69,147,159,
chr1	16857	17751	uc009vjc.1	0	-	16857	16857	0	2	198,519,
chr1	15795	18061	uc009vjd.2	0	-	15795	15795	0	5	152,159,198,136,456,
chr1	14361	19759	uc009vit.3	0	-	14361	14361	0	9	468,69,152,159,198,510,147,99,847,
chr1	14361	19759	uc009viu.3	0	-	14361	14361	0	10	468,69,152,159,198,510,147,102,54,847,
chr1	14361	19759	uc001aae.4	0	-	14361	14361	0	10	468,69,152,159,198,136,137,147,99,847,
chr1	16857	19759	uc001aai.1	0	-	16857	16857	0	6	198,136,137,147,112,847,
chr1	14361	29370	uc001aah.4	0	-	14361	14361	0	11	468,69,152,159,198,136,137,147,99,154,5
chr1	14361	29370	uc009vir.3	0	-	14361	14361	0	10	468,69,152,159,198,510,147,99,154,50,
chr1	14361	29370	uc009viq.3	0	-	14361	14361	0	7	468,152,159,198,456,154,50,
chr1	14361	29370	uc001aac.4	0	-	14361	14361	0	11	468,69,152,159,198,110,137,147,102,154,
chr1	14406	29370	uc009viv.2	0	-	14406	14406	0	7	2359,198,136,137,147,154,50,
chr1	14406	29370	uc009viw.2	0	-	14406	14406	0	7	2359,198,510,147,99,154,50,
chr1	15602	29370	uc009vix.2	0	-	15602	15602	0	7	345,159,198,136,147,154,50,
chr1	16606	29370	uc009viy.2	0	-	16606	16606	0	9	159,198,136,137,147,95,58,154,50,
chr1	16606	29370	uc009viz.2	0	-	16606	16606	0	8	159,202,136,137,147,112,154,50,
chr1	16857	29370	uc010nxs.1	0	-	16857	16857	0	8	198,136,137,147,99,227,154,50,
chr1	17232	29370	uc009vje.2	0	-	17232	17232	0	4	510,147,99,50,
chr1	17605	29370	uc009vjf.2	0	-	17605	17605	0	7	137,147,95,58,227,154,50,
chr1	16857	29961	uc009vjb.1	0	-	16857	16857	0	7	198,136,137,147,112,154,138,
chr1	34610	36081	uc001aak.3	0	-	34610	34610	0	3	564,205,361,
chr1	69090	70008	uc001aal.1	0	+	69090	70008	0	1	918,
chr1	134772	140566	uc021oeg.2	0	-	138529	139792	0	3	4924,58,492,
chr1	321083	321115	uc001aaq.2	0	+	321083	321083	0	1	32,
chr1	321145	321207	uc001aar.2	0	+	321145	321145	0	1	62,
chr1	324287	325896	uc021oeh.1	0	+	324514	325605	0	4	58,248,406,514,
chr1	322036	326938	uc009vjk.2	0	+	324342	325605	0	3	192,58,2500,
chr1	327545	328439	uc021oei.1	0	+	327745	328213	0	1	894,
chr1	323891	328581	uc001aau.3	0	+	324342	325605	0	3	169,58,4143,
chr1	367658	368597	uc010nxu.2	0	+	367658	368597	0	1	939,

1.Chrom	2.Start	3.End	4.Name	5	6
chr1	11873	14409	uc001aaa.3	0	+
chr1	11873	14409	uc010nxr.1	0	+
chr1	11873	14409	uc010nxq.1	0	+
chr1	14361	16765	uc009vis.3	0	-
chr1	16857	17751	uc009vjc.1	0	-
chr1	15795	18061	uc009vjd.2	0	-

Figure 1.3 – Galaxy's main GUI. The left panel contains the list of available tools, the right one shows the current history.

Despite having its roots in web data processing, Hadoop is sufficiently flexible to be used for a wide range of problems, including scientific data analysis. Bioinformatics applications, where datasets are often processed in large batches and do not need to be frequently updated, are particularly well-suited to MapReduce implementations [44, 61, 69]. Scientific data analysis is simplified by any tool that allows easy access to large, well-established scientific libraries, such as NumPy/SciPy [45]. However, Hadoop's native API is in Java, and the built-in library for foreign language programming[5] imposes several limitations on the available features. Pydoop[6], co-developed by the author, is a Python API for MapReduce programming and HDFS access designed to overcome most of the above limitations while allowing access to the full set of built-in and third-party Python modules [34]. Pydoop has been used to enable scalable biological data analysis in recent projects [49] and in the work presented in the following chapters.

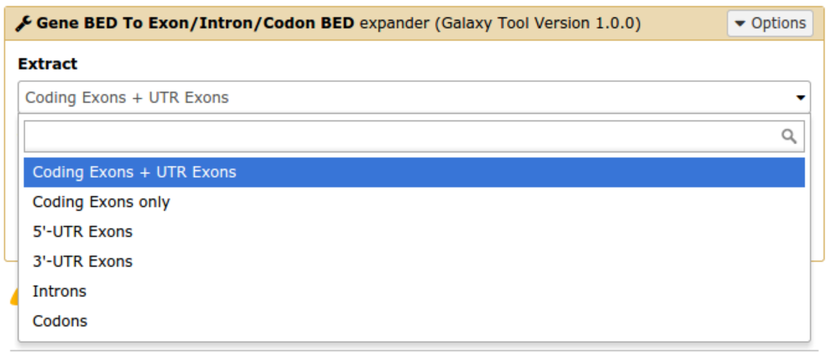


Figure 1.4 – Galaxy GUI component for a tool that extracts sub-regions from genes in BED format. The drop-down menu that allows to choose the region type is automatically generated from a *select* element in the corresponding XML wrapper.

1.3 The Galaxy Bioinformatics Platform

Galaxy[7] is an open source, web-based bioinformatics platform designed to provide easy access to a wide array of software tools, ensure reproducibility and facilitate the communication of experimental data and metadata [22]. Fig. 1.3 shows Galaxy's main user interface: the left panel displays the list of available tools while the right one shows the current history (see below). The role of the center panel is context-dependent: in this case, it shows (part of) the contents of a tabular dataset.

Galaxy simplifies access to bioinformatics applications by means of XML wrappers that provide standard abstract descriptions of input and output datasets and parameters, allowing the framework to automatically generate a GUI component for each program. The following code shows a Galaxy wrapper for a Python tool that extracts sub-regions (e.g., exons, introns) from genomic data in BED[8] format.

```
<tool id="gene2exon1" name="Gene BED To Exon/Intron/Codon BED">
<command interpreter="python">
ucsc_gene_bed_to_exon_bed.py
--input=$input1 --output=$out_file1 --region=$region "--exons"
</command>
```

[5]http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/HadoopStreaming.html
[6]http://crs4.github.io/pydoop
[7]http://galaxyproject.org
[8]http://genome.ucsc.edu/FAQ/FAQformat.html#format1

```
<inputs>
  <param name="region" type="select">
    <label>Extract</label>
    <option value="transcribed">Coding Exons + UTR Exons</option>
    <option value="coding">Coding Exons only</option>
    <option value="utr5">5'-UTR Exons</option>
    <option value="utr3">3'-UTR Exons</option>
    <option value="intron">Introns</option>
    <option value="codon">Codons</option>
  </param>
  <param name="input1" type="data" format="bed" label="from"
   help="this history item must contain a 12 field BED (see below)"/>
</inputs>
<outputs>
  <data name="out_file1" format="bed"/>
</outputs>
</tool>
```

Command line parameters are converted into graphical elements such as text boxes, buttons, etc. The `--region` option, for instance, is converted into a drop-down menu that contains all possible values (see Fig. 1.4). This model allows developers to easily extend Galaxy by writing wrappers for new tools, while the *tool shed* repository[9] provides a simple way of sharing them with the community.

In the Galaxy abstraction, a computational analysis step is performed by applying a *tool* to an input *dataset*, obtaining an output dataset. Input datasets can be created by uploading the corresponding files to the server, by importing them from established public repositories such as the UCSC Genome Browser [53], or by FTP transfer. Additionally, data can be imported from Galaxy *libraries*, server-side repositories set up by the administrator. To help the correct assignment of datasets to tool input slots, Galaxy associates a *data type* (e.g., tabular, FASTA) to each newly created dataset. Data types can be set by the user, although in many cases they are automatically inferred. Galaxy tools and datasets hide the low-level details of compute and storage management, thus mitigating the need for specialized informatics expertise when performing many common types of analysis.

Reproducibility is achieved by recording all relevant metadata about an analysis process — input and output datasets, tools and parameters — in a *history* that users can access via the right panel of the main GUI (see

[9]https://toolshed.g2.bx.psu.edu

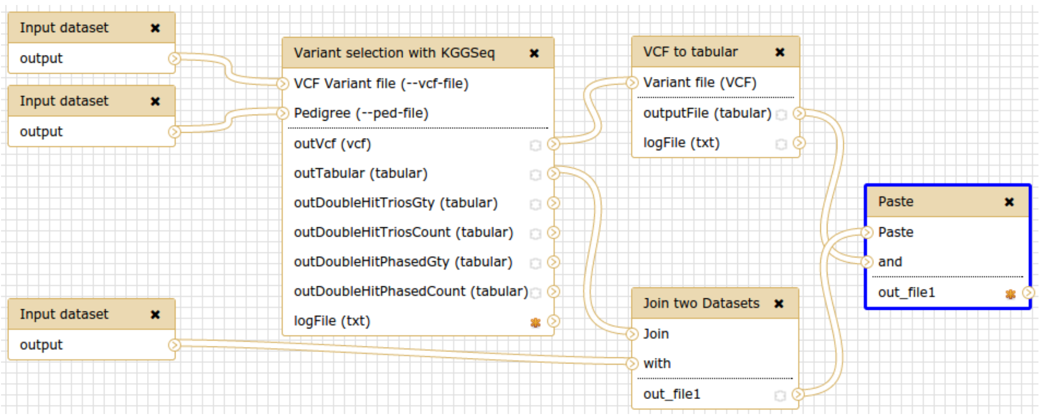


Figure 1.5 – A variant selection workflow in the Galaxy workflow editor. Example taken from the Orione repository (`https://orione.crs4.it/workflow/list_published`).

Fig. 1.3). Histories can be copied, annotated, shared and used to create *workflows*, analysis templates where several tools are chained together to produce the final output. The other main way of creating a new workflow is via Galaxy's visual editor, where tools are represented by boxes with interconnected input and output slots (see Fig. 1.5).

In this work, Galaxy is used as an accessibility layer for scalable computational tools. Moreover, annotated workflows and histories are used to ensure the reproducibility of multi-step computational transformations that lead to the creation of tracked data objects in OMERO.biobank (see Sec. 1.4).

1.4 OMERO and OMERO.biobank

The Open Microscopy Environment (OME)[10] is an open source project for biological image informatics with a focus on interoperability and traceability. The project's foundation is the OME Data Model [23], an ontology of the data types and relationships occurring in an imaging experiment that acts as a common specification for the exchange of image data and metadata. OME models images as five-dimensional (5D) structures containing multiple 2D planes with dimensions x and y, while the other dimensions represent the focal position z, the spectral channel c and the time point t. Crucially, an image stored according to the model also includes *metadata* such as details of the acquisition system, the experimental setup, and the person performing the experiment. This allows to keep the whole chain of connections between

[10] `http://www.openmicroscopy.org/`

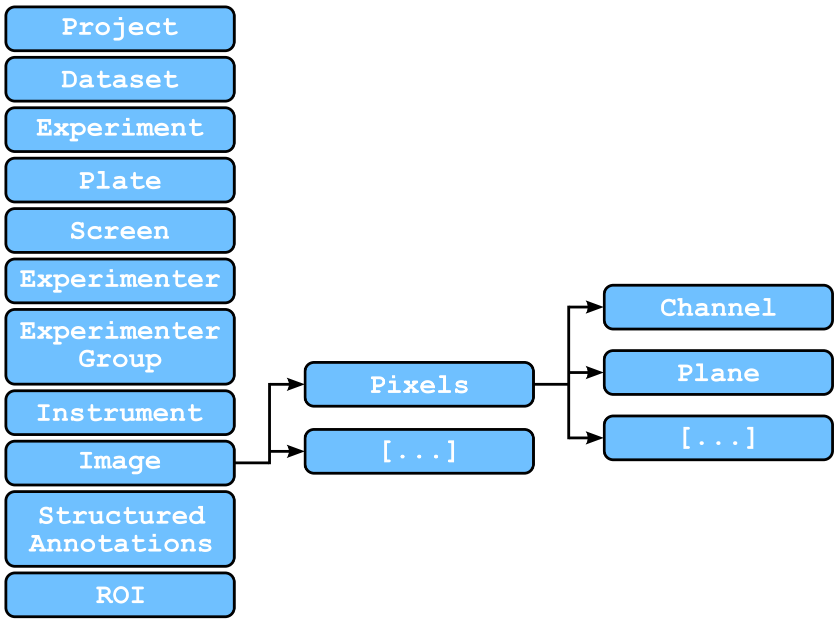


Figure 1.6 – A schematic representation of the OME Data Model. Only the top level and part of the expansion of the *image* element is shown.

the original image data and the final analysis results, which often exists only in the form of lab notes, provided it is recorded at all.

The data model is distributed as a set of XML schemas, while images are stored either as OME-XML or OME-TIFF. In the former case, one or more 5D pixel data arrays are included — compressed and encoded in base64 [27] — in an XML file that contains the metadata. OME-TIFF, on the other hand, can be seen as the dual of OME-XML: image data is stored in the widely supported TIFF format, with the metadata embedded in the file header(s). The latter solution has the advantage of making pixel data readable by any TIFF-supporting software, even though it cannot understand the metadata.

Fig. 1.6 schematically represents the top level of the current OME Data Model[11], as well as part of the expansion of the *image* element. Some entities act as containers: for instance, images are grouped in *datasets* and datasets in *projects*. An important feature of the data model is its extensibility, achieved via *structured annotations* (SAs). SAs allows individual users and organizations to add custom metadata that does not follow the core model.

[11] `http://www.openmicroscopy.org/site/support/ome-model/developers/model-overview.html`

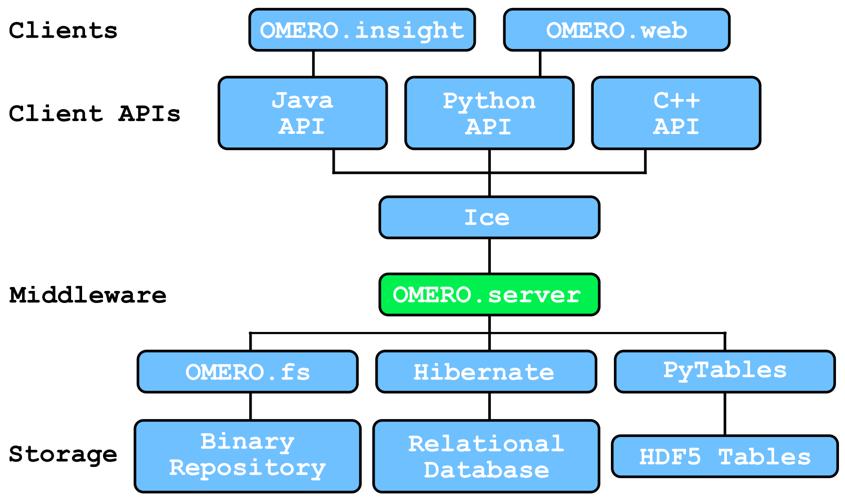


Figure 1.7 – Simplified OMERO architecture. The core server component provides access to storage and indexing resources to the client APIs via several middleware technologies.

The OME Data Model and associated file format specifications are the basis of OMERO (OME Remote Objects) [4], a software platform for biological data management that concretely provides the interoperability and provenance tracking enabled by the model itself. OMERO acts as a central repository that allows to import, archive, annotate, visualize and export images, and supports the execution of analysis scripts written in Python.

Fig. 1.7 shows a simplified diagram of OMERO's architecture. A core server application, *OMERO.server*, connects a series of data repositories to the clients via several middleware components: *OMERO.fs* manages access to the binary repository, where actual image data is stored; Hibernate [46] is used to mediate interactions with the RDBMS that stores metadata; PyTables[12] allows to manipulate HDF5 [18] tabular data; Ice[13] provides a unified, multi-language API for clients, either the ones included in the OMERO distribution or third-party ones. Interoperability between the different file formats is achieved via Bio-Formats [35], a Java library that extracts both data and metadata from a large number of image formats and converts them to OME-TIFF.

OMERO.biobank[14], co-developed by the author, is a software platform for data-intensive biomedical research based upon the core services of OMERO. Fig.1.8 schematically shows OMERO.biobank's architecture. The core com-

[12]https://pytables.github.io
[13]https://zeroc.com
[14]http://www.openmicroscopy.org/site/support/partner/omero.biobank

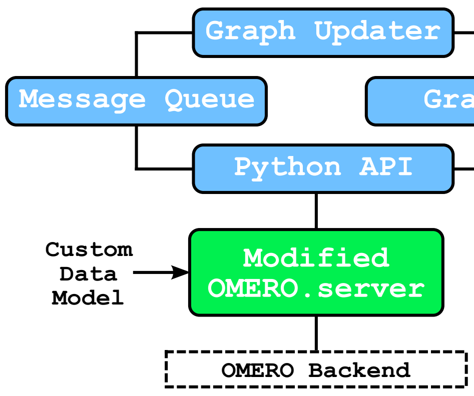


Figure 1.8 – Simplified OMERO.biobank architecture. The core component is a modified version of OMERO.server that uses custom models tailored to data-intensive genomics.

ponent is a modified version of OMERO.server, based on a custom data model that defines *entities* — such as laboratory samples, genomics datasets and analysis results — connected by *actions* that keep track of provenance information. To speed up the management of the complex chains of connections between objects, entities are mapped to nodes and actions to edges in a graph database (currently built on Neo4j[15]). Synchronization between OMERO objects and the graph DB is ensured by a message queuing system (currently implemented with RabbitMQ[16]) that propagates relevant transactions to an update service. The storage backend and related middleware, summarized as "OMERO backend" in the diagram, has the same architecture as in the original OMERO. The RDBMS structure, however, is different, since it is generated from the custom models; the same holds for the HDF5 tables, which are OMERO.biobank-specific and store genomic data and annotations.

Fig. 1.9 shows a small, simplified section of the custom data model. The *study* object, representing an entire research project, is the root object that provides a common context under which other objects are defined. All *individuals* participating in a study are linked to it via an *enrollment*, and to other individuals via kinship relationships. An *action* is an abstraction for any process, either physical or computational, that links an entity to the one that generated it. Walking back the chain of actions and entities that generate them allows to reconstruct provenance information; in practice, for efficiency reasons, this is done on the graph DB mapping discussed above.

[15]http://neo4j.com
[16]https://www.rabbitmq.com

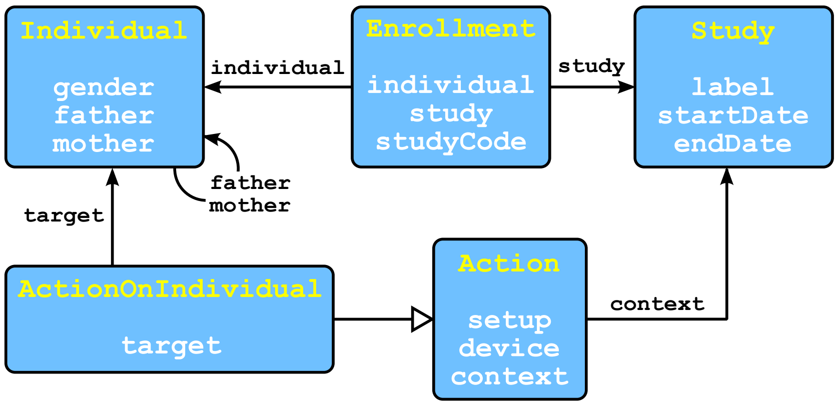


Figure 1.9 – A simplified section of the OMERO.biobank data model. Boxes represent entities, with attributes listed in white. The line ending with the empty arrowhead represents inheritance, while the other ones indicate relationships.

Actions are carried out by *devices*, generic actors that can create new entities. A device can be a physical object, such as an acquisition machine, a software program (e.g., a bioinformatics tool that produces analysis results from an input dataset, or an import utility that creates a new dataset from outside the system) or even a whole analysis pipeline. An example of the latter representation is given by Galaxy workflows (see Sec. 1.3), which can be exported from a Galaxy instance as a JavaScript Object Notation (JSON)[17] record, stored as metadata in an OMERO.biobank device and re-imported to launch a set of computations on a given data object.

1.5 Enabling Data-Intensive Biomedical Studies

Current large-scale biomedical research projects are often characterized by one or more of the complex issues discussed above, mostly related to the size and heterogeneity of the datasets involved. In contrast with what happened in the past, such projects cannot be carried forward with the sole aid of lab notebooks and off-the-shelf software, but require ad-hoc computing infrastructure capable of providing adequate scalability, reproducibility and traceability. This work presents a methodology — based upon the state-of-the-art technologies described in the above sections — to build such an infrastructure, through several examples of computational solutions that have

[17]http://json.org

been employed to tackle specific research problems.

The following papers have been published as a result of the work presented here:

- Simone Leo, Luca Pireddu, and Gianluigi Zanetti. "SNP genotype calling with MapReduce". In: *Proceedings of the Third International Workshop on MapReduce and Its Applications*. 2012, pp. 49–56. DOI: `10.1145/2287016.2287026` (chapter 2);
- Andrea Calabria, Simone Leo, Fabrizio Benedicenti, et al. "VISPA: a computational pipeline for the identification and analysis of genomic vector integration sites". *Genome Medicine* 6 (9), 2014. DOI: `10.1186/s13073-014-0067-5` (chapter 2, chapter 3; Calabria and Leo are equal contributors);
- Simone Leo, Luca Pireddu, Gianmauro Cuccuru, et al. "BioBlend.objects: metacomputing with Galaxy". *Bioinformatics* 30 (19), 2014, pp. 2816–2817. DOI: `10.1093/bioinformatics/btu386` (chapter 3).

In addition, the work published in Calabria, Leo et al. has been used to enable viral integration site analysis in the context of the study published in the following paper:

- Alessandra Biffi, Eugenio Montini, Laura Lorioli, et al. "Lentiviral hematopoietic stem cell gene therapy benefits metachromatic leukodystrophy". *Science* 341 (6148), 2013. DOI: `10.1126/science.1233158` (Calabria and Leo are co-authors).

Chapter 2

Enabling Scalability in Biomedical Applications

2.1 Scalable Genotype Calling

DNA sequence variations are widely studied to identify their influence on phenotypic traits such as height, longevity and susceptibility to specific diseases. In the simplest case, a disorder is due to a mutation in a single gene (*monogenic* disease); often, however, it is the result of numerous variants acting in concert (*polygenic* or *complex* disease). In order to uncover the genetic basis of complex disorders, a large number of variants must be studied simultaneously.

The availability of high-density genotyping microarrays [29] has paved the way for genome-wide association studies (GWAS) [13], where hundreds of thousands of variants are measured simultaneously across the genome to identify risk factors for diseases. For instance, the Affymetrix Genome-Wide Human SNP Array 6.0 [39], one of the most widely used genotyping platforms, simultaneously measures over 900,000 variants across the human genome.

The estimation of genetic variants from the raw data gathered via genotyping arrays is known as *genotype calling* (GC). GC on GWAS data collected with Affymetrix platforms involves the analysis of a large number of samples, with results depending on the overall statistics, the quality of each sample, and the number of samples processed concurrently. The latter factor is particularly important, since processing data in small batches leads to well-known adverse effects [31, 50]. In the case of an ethnically homogeneous population, the batch size required to counteract these effects can be as large as the total number of available samples.

The reference GC algorithm for Affymetrix data analysis, Birdseed [28], is implemented by the `apt-probeset-genotype` program in the Affymetrix Power Tools (APT)[1]. For a large-scale GWAS, where the number of samples can be in the order of thousands or more, `apt-probeset-genotype` faces serious scalability issues, such that completing the GC analysis would take months even on a fast computational node. This makes one-shot analysis barely feasible, and systematic studies on how the results depend on the statistics of the samples in the dataset virtually impossible. Although the program allows a certain degree of load distribution — achieved by processing a subset of the variants at a time — this approach is inefficient, because the initial normalization procedure on which it depends must be re-executed for every subset. For this reason, the application is typically run on relatively small batches of samples, leading to the adverse effects discussed earlier.

This section describes a MapReduce (see Sec. 1.2) implementation of GC analysis for Affymetrix arrays that overcomes the aforementioned scalability limitations by distributing the workload across many computing nodes in a cluster. The application reproduces the core functionality of the APT version while enabling the concurrent analysis of thousands of samples in the same batch, thus avoiding the adverse effects described above.

2.1.1 Background

Single nucleotide polymorphisms (SNPs) are variations at a single genomic position. Usually, a SNP consists of two possible variants (*alleles*), conventionally denoted by the letters A and B. Thus, diploid organisms (including humans) can have one of three possible genotypes at each SNP site: AA, AB and BB. In GWAS, a large number of SNPs distributed across the whole genome are tested for correlation with qualitative (e.g., a disease) or quantitative (e.g., body mass index) phenotypic traits.

Although several different techniques have been developed for measuring genotypes, this section concentrates on high-throughput genotyping with SNP microarrays such as the aforementioned Affymetrix 6.0. Genotyping arrays are based on the biochemical binding of complementary nucleotides: A to T and C to G. The array surface is covered with probes, each complementary to a DNA sequence harboring a particular SNP. Probes are included for each variant (A or B) and are grouped in redundant collections called *probesets*. In a genotyping run, DNA is fragmented, amplified via polymerase chain reaction (PCR), labeled with a fluorescent dye and hybridized to the probes. The intensity of the fluorescence at each probe location is then measured,

[1] http://www.affymetrix.com/partners_programs/programs/developer/tools/powertools.affx

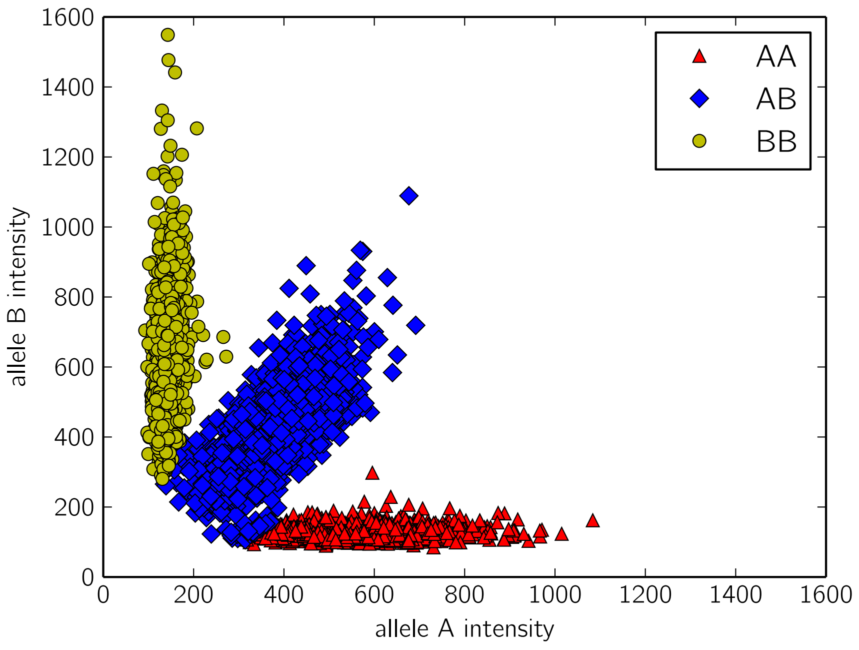


Figure 2.1 – Fluorescence intensities of a probeset in 5,722 samples. Birdseed classifies each sample as homozygous for the A allele (AA cluster), homozygous for the B allele (BB cluster) or heterozygous (AB cluster).

and its value is stored in an Affymetrix CEL file.

As mentioned earlier, the standard processing pipeline for Affymetrix microarrays is `apt-probeset-genotype`: raw probe intensities from CEL files are normalized and summarized (i.e., converted to per-probeset values), then GC is performed by fitting a two-dimensional Gaussian mixture model (GMM) to the resulting dataset, where the two dimensions represent the summarized intensities for allele A and B. The fitting is done by expectation-maximization initialized with prior expected statistics obtained from HapMap [63] data. For each probeset, the model determines the AA, AB and BB clusters (Fig. 2.1) and assigns each sample to one of the three corresponding classes, while the distance from the cluster's centroid provides a measure of the level of confidence associated with the call. A limit may be placed on the confidence level, beyond which the SNP is classified as a *no call*.

Since SNPs on the X and Y chromosomes must be treated differently according to the gender of the sample, and a-priori gender information can often be unknown or unreliable, APT includes a tool for estimating it from the probe intensities (gender calling). This estimation can be done using either one of two methods, one based on the relative intensities of probe signals on the Y and X chromosome, and the other on an expectation-maximization algorithm.

The normalization step is a key performance-influencing factor, since data from all input samples must be processed at the same time. The `apt-probeset-genotype` tool employs a quantile normalization [9] procedure that equalizes the distribution of probe intensities across samples. Let m be the number of probes, n the number of samples, and A the $m \times n$ matrix whose columns represent the probe intensity vectors:

$$A_{ij} = i(p_j, s_k) \qquad (j = 1, \ldots, m, \quad k = 1, \ldots, n)$$

where $i(p_j, s_k)$ is the intensity of probe j in sample k. Quantile normalization acts as follows:

1. each column in A is sorted;
2. in each row, all elements are replaced by the row mean;
3. each column is rearranged to have the same ordering as the original A.

In large-scale GWAS, conventional single-core implementations fail to perform the above computation efficiently, since the normalization matrix becomes too large to be kept in memory, hence the motivation to develop a distributed version of the algorithm.

2.1.2 MapReduce Implementation

The main difficulty in implementing a distributed version of the GC workflow stems from the fact that the dataset has to be traversed along either the samples or the probes dimension according to the specific step being performed (note that this is trivial in a single-processor implementation, where all data can be kept in the same array). Due to this requirement, the procedure has been implemented as a chain of Pydoop (see Sec. 1.2) jobs, transposing the ideal sample-probeset matrix as needed and storing intermediate results to HDFS after serializing them with protocol buffers[2]. Although most of the code has been written from the ground up in Python, it includes two extensions modules built as Boost.Python [1] wrappers for sections of the APT code: `algo`, which contains gender calling and GMM fitting, and `io`, which wraps routines for reading CEL and chip layout files. Individual steps are described in the following paragraphs, while the overall workflow is schematized in Fig. 2.2.

[2] https://developers.google.com/protocol-buffers

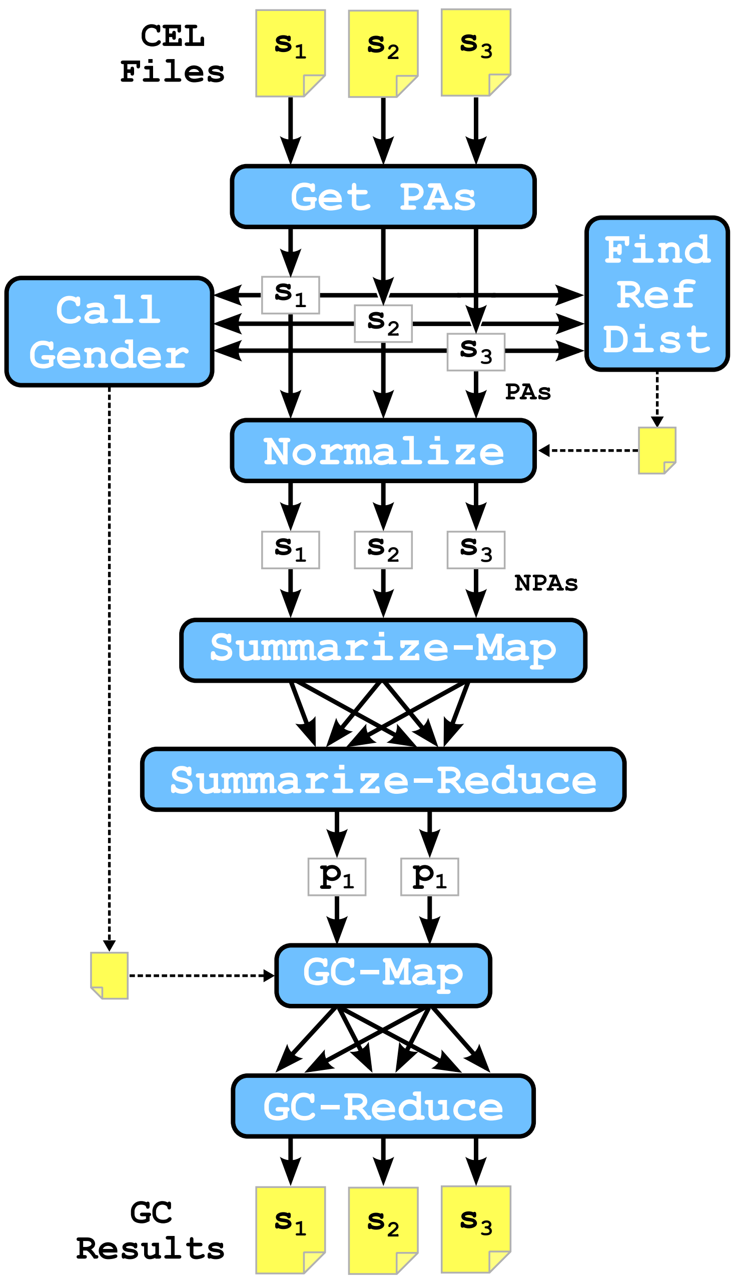


Figure 2.2 – The MapReduce GC workflow. Initial data from the microarray assays is stored by sample in Affymetrix CEL files. “Get PAs” and “Normalize” are map-only applications that process data by sample, while “Summarize” and “GC” use the full MapReduce paradigm to transpose data according to the required processing direction. “Call Gender” and “Find Reference Distribution” store auxiliary data needed by subsequent steps to HDFS.

Get Probe Arrays

This is a map-only stage that, for each CEL file stored in HDFS, extracts probe intensities and stores them in a "probe array" (PA) object. Each PA is then serialized and written back to HDFS. Since multiple subsequent steps take their input from probe intensities, storing PAs allows to execute the data extraction process only once for the entire workflow.

Call Gender

This map-only step deserializes PAs read from HDFS and performs gender calling as described above. The local wrapper script that launches the Hadoop job collects gender estimations for all samples in a tab-separated HDFS file for later usage by the GC step.

Find Reference Distribution

This stage computes the mean quantile distribution of probe intensity vectors. As described in Sec. 2.1.1, this is achieved by taking the row averages of the matrix whose columns are the sorted intensity vectors. To implement this procedure in MapReduce, the mapper, for each PA, emits a key/value pair structured as follows: the key is the host name of the cluster node where the map task is running, while the value is a structure consisting of the sorted intensity vector and a sample counter set to 1. The reducer sums all vectors and counters and emits them using the same structure used by the mapper. Finally, a local script (in practice the same one that launches the Hadoop job) computes the overall vector sum and divides by the counter to get the average intensity vector, which is written to HDFS for subsequent usage by the normalization step.

Using the host name as the intermediate key allows to distribute the workload by host so that the final sum, which involves at most a number of vectors equal to the size of the cluster, can easily be handled by a local script. A more fine-grained distribution could be achieved by having each mapper sort the PAs and emit key/value pairs consisting of row numbers and the corresponding intensity values. In this case, the reducers would sum all the values for each row index and emit the index itself and the sum as the output key/value pair, leaving it to the local script to reconstruct the final vector from the single elements and indices. This arrangement would allow the computation to scale with respect to the number of probes in a microarray. In practice, however, since current machines can easily hold a single probe intensity vector in memory, this is not necessary and would actually lead to worse performance due to the much higher number of intermediate values transmitted over the network. At the other extreme, a solution where PA vectors are emitted to a single reducer would eliminate the need to post-process partial sums with a local script, but also introduce a bottleneck that would greatly reduce scalability, hence the adopted hybrid solution.

Normalize

This map-only step converts PAs to normalized PAs (NPAs) using the reference distribution computed in the previous stage. As described in Sec. 2.1.1, each probe intensity value is replaced by the corresponding one in the reference distribution.

Summarize

This stage summarizes NPAs, yielding per-probeset intensity values. Up to this point, data has been processed by sample: each HDFS file stored a serialized data structure holding all probe intensities for a specific sample. In order to perform the clustering described above, however, the subsequent GC step needs to process data by probeset. For this reason, this step uses the MapReduce framework to transpose the dataset while performing the summarization. For each probeset, the mapper stores all intensities of probes belonging to that set in a vector, and emits a key/value pair with the probeset ID as the key and the vector itself as the value. The reducer, via the `algo` extension, uses the PLIER routine from the APT libraries to summarize each vector to a per-probeset scalar value. Since there is one such value for each input sample, the end result is again a series of vectors, which are serialized and stored to HDFS as in previous steps.

Genotype Call (GC)

This is the final stage, where actual SNP calling is performed. The mapper reads summarized data from the previous step and applies the clustering algorithm (made available via the `algo` extension) to call genotypes for each probeset. Since final results must be regrouped by sample, the mapper emits call outputs using the sample ID as the intermediate key, so that the reducer can write one output file per sample.

2.1.3 Evaluation

Accuracy

The accuracy of the MapReduce workflow has been evaluated by comparing its results with those obtained by the APT version for a 6863 samples dataset analyzed in the course of a previous study [65]. The MapReduce version achieved a slightly lower average *no call* rate (5.017×10^{-2} vs 5.206×10^{-2}, using the default confidence threshold of 0.1), so that approximately twelve million additional genotypes were called over all samples. The average allelic discordance rate (i.e., the fraction of single allele mismatches between the two implementations) was 3.6×10^{-4} with a standard deviation of 10^{-4}, which is lower than the genotyping chip's intrinsic error rate of 3×10^{-3} (measured as discordance with respect to the HapMap genotypes[3]).

Scalability

The performance of the MapReduce workflow has been evaluated both by comparison with that of `apt-probeset-genotype` and by measuring its scalability with respect to the number of samples and the size of the cluster. Tests have been executed on a cluster of machines running CentOS Linux version 5.2 and configured as follows: two quad-core, 2.8 GHz CPUs; 16 GB of RAM; two 250 GB hard disks, only one of which used for HDFS storage (the other one being exclusively used by the operating system); 1 Gb Ethernet network interface. Machine sets used to run Hadoop have been configured with two nodes dedicated, respectively, to the HDFS and MapReduce master, while the other ones hosted an HDFS and a MapReduce slave each. The test dataset consisted of 7292 CEL files produced in the course of a study on autoimmune diseases [54].

In the discussion that follows, scalability will be evaluated with respect to the normalized throughput, defined as the number of samples processed per time unit by each computing node (on average). Let N be the number of cluster nodes, S that of input samples and Δt the total running time. The normalized throughput is then given by:

$$T = \frac{S}{N\Delta t}$$

In the ideal case (*linear speedup*), as the number of nodes increases, the throughput $S/\Delta t$ increases by the same amount, so that T remains constant. In practice, speedup is often sub-linear due to several factors, most notably

[3]http://www.affymetrix.com/support/technical/datasheets/genomewide_snp6_datasheet.pdf

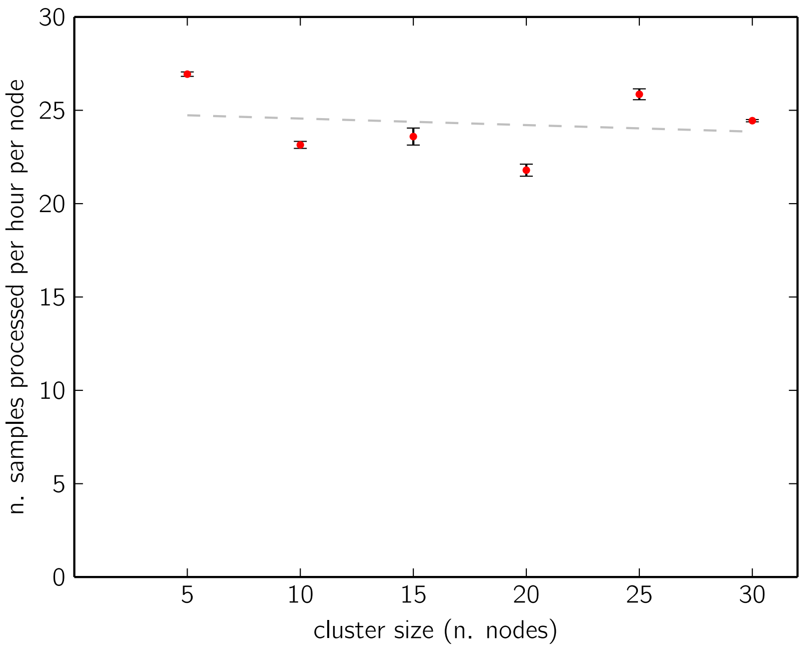


Figure 2.3 – Normalized throughput of the MapReduce workflow on a 1823 samples dataset. The dashed line represents the linear least squares fitting.

the presence of non-distributed sections (in this case, the local scripts described above) and network communication overhead (more pronounced in full MapReduce applications than in map-only ones).

Fig. 2.3 shows the normalized throughput of the MapReduce implementation, run on a 1823 samples dataset, for varying cluster sizes. Data points are mean values (the corresponding error bars are shown in black) computed over three iterations of the same run, while the dashed line represents the linear least squares fitting. In this size interval, the application maintains good scalability, with only a slight deviation from the ideal flat line.

Fig. 2.4 shows the normalized throughput of the MapReduce implementation, run on a 30-node Hadoop cluster, for varying dataset sizes. In this case, since the independent variable is the input size, the ideal behavior depends on the computational complexity of the various components, including those controlled by the Hadoop framework. In particular, since all stages that include a reduce phase give rise to a comparison-based sort, the complexity is at least linearithmic. In addition, even though their job is mostly limited to data collection, local scripts take more time as the number of objects they have to manipulate increases.

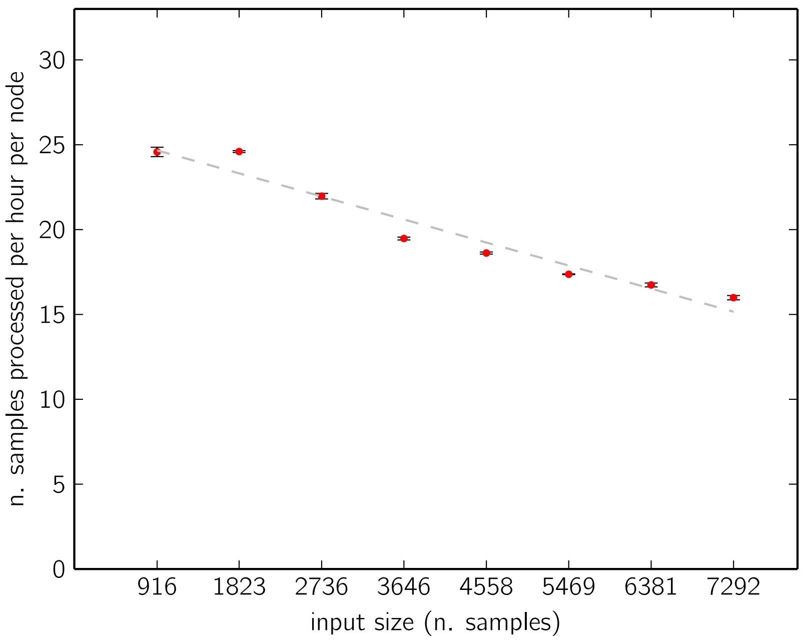


Figure 2.4 – Normalized throughput of the MapReduce workflow on a 30-node cluster. The dashed line represents the linear least squares fitting.

Fig. 2.5 shows the total running time for the APT baseline and the MapReduce implementation running on a 30-node Hadoop cluster. The former, due to its non-distributed nature, is characterized by a rapidly growing running time, which reaches the order of several days even for relatively modest dataset sizes; the latter, on the other hand, requires less than one day even when the full test dataset is used.

In a previous attempt at distributing the workload without changing the implementation, Valentini et al. [65] partitioned the data along the probeset dimension and ran a separate `apt-probeset-genotype` instance for each partition. However, since the normalization matrix must be recomputed for each input partition, this strategy does not scale well: GC on 6836 samples was completed in 15 days on 18 nodes ($T = 1.1$). In the same work, the authors were able to achieve performance comparable to that of the MapReduce version ($T = 20.4$) by partitioning the data by sample into seven batches. In this case, however, each batch is normalized separately, leading to the adverse effects discussed earlier. To minimize such effects, batches were carefully balanced with respect to variables such as the ratio of cases and controls and plate and laboratory representation, a lengthy procedure that counterbal-

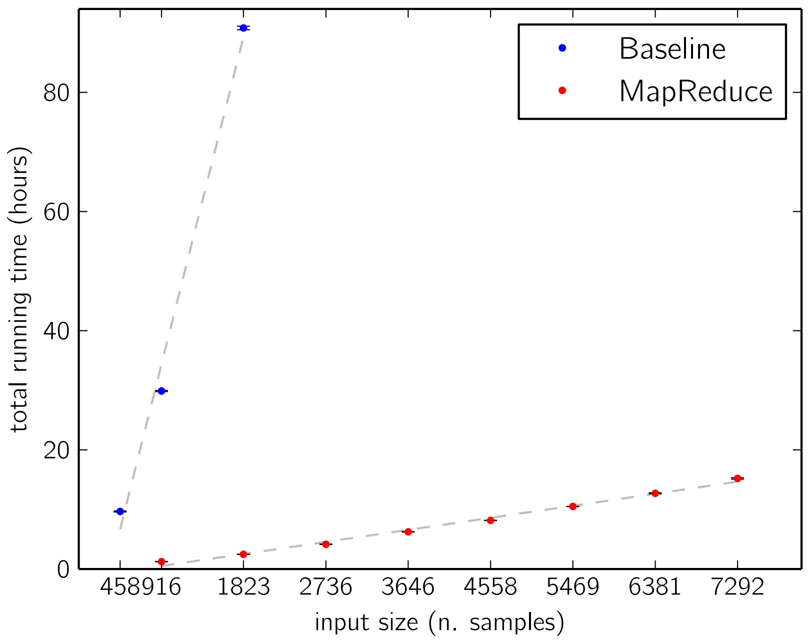


Figure 2.5 – Total running time as a function of the number of samples, for the APT baseline and the MapReduce workflow running on 30 nodes.

ances the speed gain obtained in the subsequent computation. Moreover, even after balancing, results were different from those of the by-probeset partition case (where all samples are used for normalization), with an average allelic discordance rate of 3.4×10^{-3}. In addition, this strategy is both time-consuming and error-prone, not only due to the aforementioned group balancing, but also because each individual job must be manually launched, monitored and rerun in case of failure, whereas in the Hadoop version these aspects are taken care of by the framework.

2.2 Scalable Viral IS Analysis

Gene therapy (GT) involves the delivery of a therapeutic DNA sequence into diseased cells by means of appropriate *vectors*. Viruses, due to their ability to integrate their genome into a host cell, are commonly used for this purpose. In particular, γ-retroviral (γ-RV) and lentiviral (LV) vectors have been successfully used in hematopoietic stem cell GT (HSC-GT) [2, 3, 7, 10].

A major concern with HSC-GT is the potential harmful alteration of gene expression in the neighborhood of the vector's integration site (IS), known as *insertional mutagenesis* (IM) [12, 24, 47, 51]. Consequently, analyzing the genomic distribution of ISs in the cells of GT patients is of critical importance to assess the safety of the therapy. Moreover, IS datasets can be used to quantify clonal diversity, allowing to assess the efficacy of hematopoietic reconstitution [7].

Viral IS analysis starts with the amplification, via polymerase chain reaction (PCR), of the region where the cellular genome flanks the provirus, followed by sequencing and bioinformatics processing. In the recent past, the wet lab part of this procedure has seen consistent advances, mostly due to the availability of NGS machines and to the introduction of new cell types and time points. The computational part, on the other hand, still poses several major challenges, most notably the efficient analysis of the huge amount of data produced by NGS platforms — the most intensive step being the alignment of sequencing reads to the host's reference genome.

This section describes a bioinformatics pipeline for IS analysis that uses Hadoop to distribute the sequence alignment workload across a cluster of computing nodes. The focus here is on scalable sequence alignment, while accessibility and reproducibility will be discussed in the next chapter. The pipeline has been successfully employed in clinical studies on metachromatic leukodystrophy (MLD) [7], Wiskott–Aldrich syndrome (WAS) [2] and globoid cell leukodystrophy (GLD) [30]. The software has been published as open source and is available at `https://github.com/crs4/vispa`.

2.2.1 Pipeline Structure

The bioinformatics pipeline targets DNA reads generated by sequencing vector-genomic junctions that have undergone linear amplification-mediated PCR (LAM-PCR) [55]. In these sequences, genomic fragments are flanked by the viral long terminal repeat (LTR) and a linker cassette (LC) [42]. Often, to achieve the full utilization of the sequencer's capacity, multiple DNA samples are pooled together and processed in the same run, a technique known as *multiplexing*. To enable the subsequent identification of individual samples from the output reads, a specific *barcode* sequence is added to each sample before pooling. All these non-genomic sequence fragments must be removed from the reads before the alignment phase.

The rest of this subsection outlines the pipeline structure to provide the context required for discussing the distributed sequence alignment and filtering application. Additional information on the various tools is provided in the next chapter.

Format Conversion

Subsequent steps expect data in the ubiquitous FASTA format[4]. NGS data, however, comes in several different formats according to the specific technology used. The first step in the pipeline is therefore a format conversion utility. Currently, it supports the standard flowgram format[5] (SFF) used by 454 sequencers[6].

Demultiplexing

This step reads the stream of FASTA sequences output by the previous one, assigns each sequence to a sample according to the leading barcode and writes an output file for each sample. The tool, implemented in Python, takes the list of known barcodes as additional input, and discard all reads for which no match is found.

Trimming

In this step, the LTR and the LC are removed from the reads to isolate the genomic portion. The application employs a Boost.Python [1] wrapper for the NCBI C++ Toolkit[7] (`https://github.com/crs4/blast-python`) to search for the last 63 nucleotides of the LTR. If found, the LTR is trimmed out and the resulting sequence is kept for further analysis; otherwise, since the absence of the LTR is a sign of an aspecific amplification product, the read is discarded. The same is done for the LC, with the difference that non-matching reads are not discarded.

Alignment and Filtering

In this stage, trimmed reads are aligned to a reference genome as a first step to determine the viral IS. Since the latter is defined as the junction between the integrated vector and the host genome, the sequence fragment adjacent to the LTR has to be identified as accurately as possible. For this reason, sequence mapping is followed by the application of a series of filters aimed at improving the quality of alignment hits [6, 67]:

1. the match must begin within 3 bp of the LTR end (although 3 is the recommended choice, the parameter is configurable by the user);

[4]`http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml`
[5]`http://www.ncbi.nlm.nih.gov/Traces/trace.cgi?view=doc_formats`
[6]`http://www.454.com`
[7]`http://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC`

2. the identity score (percentage of matches between the read and the genome) must be higher than a specified threshold;
3. the alignment must be *unambiguous* (i.e., the read must not map to multiple genomic regions with hits of comparable quality).

The latter condition is satisfied as follows. For every mapping m of a given read i, consider the following definition of *homology score*:

$$h_{im} = 100 \, \frac{|q_s - q_e|_{im}}{l_i}$$

where q_s, q_e are, respectively, the starting and ending position of the input read in the alignment (reported as “query start” and “query end” by BLAST) and l is the length of the read itself. A read is classified as unambiguously mapped if both its alignment score s (as reported by BLAST) and its homology score h (as defined above) are significantly better than those of the second best hit:

1. sort all mappings for a given read by decreasing alignment score:

   $$s_{(1)}, s_{(2)}, \ldots$$

2. classify the read as unambiguously aligned if:

   $$|s_{(2)} - s_{(1)}| > s_t \quad \wedge \quad |h_{(2)} - h_{(1)}| > h_t$$

   where s_t, h_t are predefined thresholds.

In the course of the studies cited above, after a parameter tuning process performed on a controlled dataset, s_t and h_t were set, respectively, to 15 and 20 (these parameters are also configurable in the implementation).

Sequence alignment is a computationally intensive operation that can take a very long time if executed on a single machine. In the course of the clinical trials cited above, for which the pipeline was developed, the input dataset consisted of nearly fourteen million reads, and the time required to align them on a single processor was estimated to be in the order of the years. For this reason, this step is implemented as a distributed sequence alignment and filtering application (see Sec. 2.2.2), based on a Pydoop wrapper for BLAST [5].

IS Merging

Due to technical biases, the position of a given IS can lie in a 3 bp interval around the start of the alignment. This step takes this variability into account by merging together all ISs that lie in the same 3 bp window, effectively considering them as manifestations of the same integration event.

IS Annotation

Finally, ISs are annotated by listing nearby genomic features (e.g., genes, miRNAs). Besides the list of merged ISs determined earlier, the program takes as input an annotation file containing a list of known features (such as the ones available from the UCSC Genome Browser[8]) and, for each IS, finds the closest ones.

2.2.2 Distributed Alignment and Filtering

Fig. 2.6 schematizes the architecture of the distributed sequence alignment and filtering application. Implemented as a map-only Pydoop (see Sec. 1.2) job, the tool parallelizes execution by query sequence: each map task receives a subset of the input reads, calls BLAST in a subprocess and classifies alignments as described in Sec. 2.2.1. In practice, this is done by adding an appropriate tag to each sequence ID in the MapReduce output; a local collector script subsequently partitions alignments into different output files according to their tag. Matching sequences are classified as unambiguously aligned if they meet the above requirements; otherwise they are classified as *repeats*. In addition, non-matching sequences are tagged as such, leading to three output files written by the collector.

Since FASTA files are characterized by multi-line records, Hadoop's standard input format, which treats each input line as a record, cannot be used (a FASTA entry might be split among two different map task, leading to parsing errors or wrong sequences). For this reason, the application uses a custom record reader that splits FASTA files into individual entries. In practice, to decouple record reading and alignment, the former is performed by a separate map-only job that converts records to a tab-separated format suitable for processing by the default Hadoop reader.

[8]http://genome.ucsc.edu

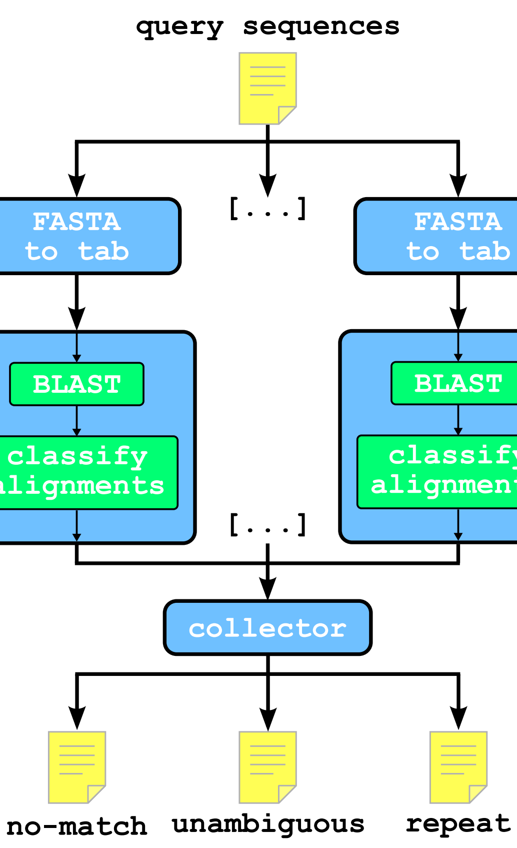


Figure 2.6 – Schematic representation of the distributed sequence alignment and filtering application, implemented as a map-only Pydoop job. Each mapper runs a Python wrapper for the BLAST executable and filters results as described above. Finally, a local script collects individual outputs and writes the final results to a file.

2.2.3 Evaluation

The pipeline's accuracy has been evaluated both by comparison with other IS analysis software on an *in silico* dataset, and by analyzing real-world datasets from [7] and [2]. Details of this evaluation, mainly performed by co-authors A. Calabria and G. Spinozzi, are available from the VISPA paper [14], while this subsection concentrates on the scalability of the distributed part.

Fig. 2.7 shows the normalized throughput (see Sec. 2.1.3) of the distributed alignment and filtering application on a synthetic dataset of 2048 64-base sequences, for varying cluster sizes. Deviations from the ideal flat line can be ascribed to several factors, mostly connected to the fact that the test dataset has been kept relatively small to allow for multiple iterations.

Hadoop distributes the workload by assigning the same number of input

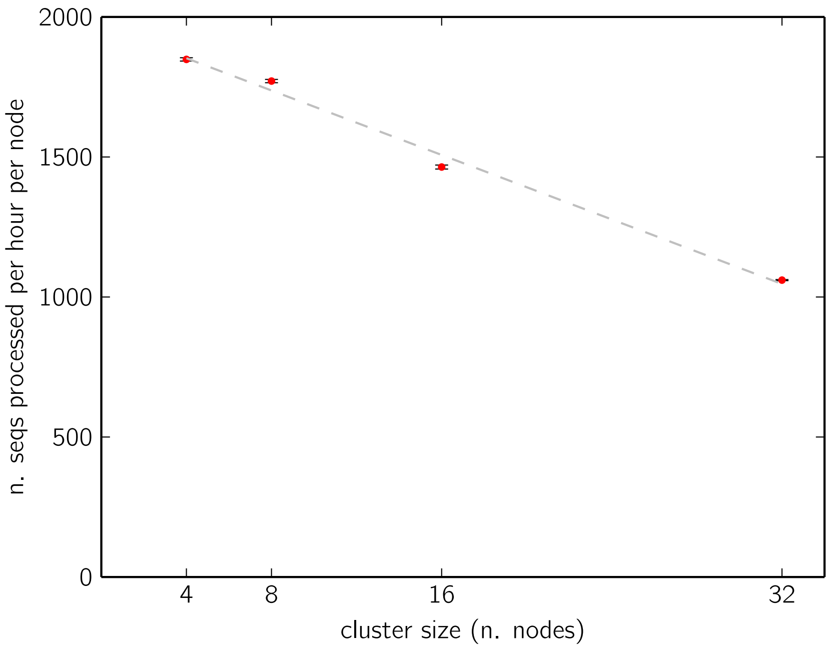


Figure 2.7 – Normalized throughput of the distributed alignment and filtering application on a synthetic dataset of 2048 64-base sequences. The dashed line represents the linear least squares fitting.

lines (and thus of input sequences, due to the tabular format used) to each map task. This simple and usually very effective strategy does not lead to optimal results in this case, even though all reads have been generated with the same length. The main reason for this is that the time required to process a query sequence depends on the number of matches with the database (the reference genome, in this case). BLAST starts its search with a *seeding* phase, looking for high-scoring (according to an appropriate substitution matrix) *word* pairs between the query and the target sequence (a word is a short subsequence of fixed length). Only word pairs whose score exceeds a predefined threshold are kept for the subsequent *extension* phase, where the initial seed is stretched to yield the full alignment. Query sequences with more high-scoring pairs (HSPs) give rise to more extensions and thus longer execution time.

The variable processing time of input records leads to imperfect load balancing and thus to suboptimal scalability. Since the difference is tied to the degree of similarity between the query sequence and the database, which is part of what the algorithm itself has to find, devising a custom

Hadoop input format capable of obtaining a better load balance is far from trivial. If the number of sequences processed by each map task is sufficiently large, however, it is more likely that they will have a similar processing time. Consequently, a small dataset has a greater chance of leading to bad scalability.

Chapter 3

Enabling Reproducibility in Biomedical Applications

3.1 Reproducible Viral IS Analysis

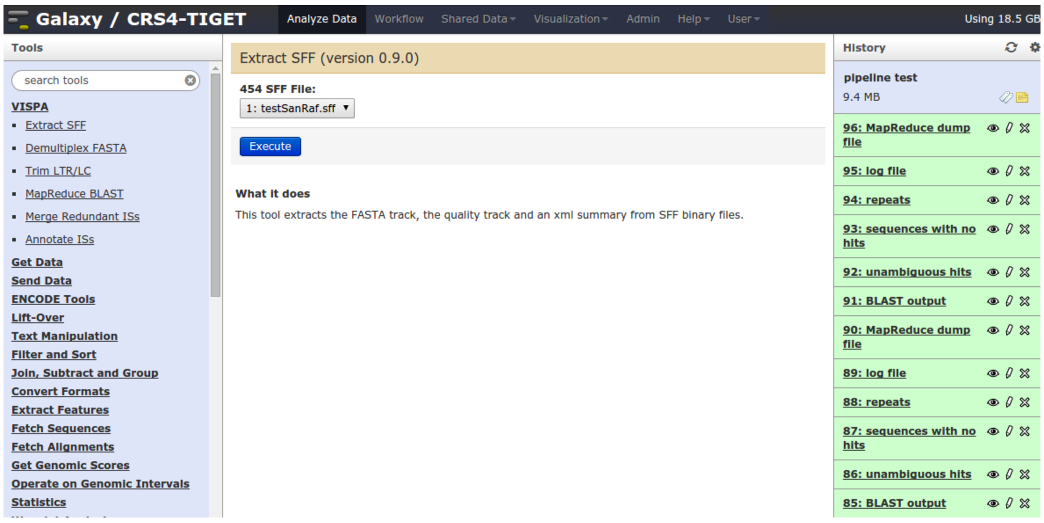
Sec. 2.2.1 describes a pipeline for viral IS analysis that consists of multiple interconnected steps, each characterized by different data types and parameters. Although all developed tools can be manually installed and used via their command line interfaces (CLIs), this mode of operation is typically only adequate for users with a computer science background. Moreover, due to the pipeline's complexity, the reproducibility of analysis results must be ensured by an automated mechanism.

As discussed in Sec. 1.3, Galaxy provides a common framework that allows to make computational tools accessible to life science personnel with various degrees of technical knowledge, while taking care of reproducibility via the history mechanism. All tools introduced in Sec. 2.2.1 have been integrated into Galaxy as described below. Fig. 3.1 shows the custom Galaxy instance used for IS analysis in the course of the clinical trials [2, 7, 30]. The expansion of the VISPA sub-menu is shown at the top of the left panel, while the center one displays the first format conversion tool.

Format Conversion

This tool has been implemented as a wrapper for `sff_extract`[1] (later merged into `seq_crumbs`[2]). This tool takes a 454 SFF file as input (selectable via the drop-down menu shown in Fig. 3.1) and produces four output files containing,

[1]http://bioinf.comav.upv.es/sff_extract
[2]http://bioinf.comav.upv.es/seq_crumbs



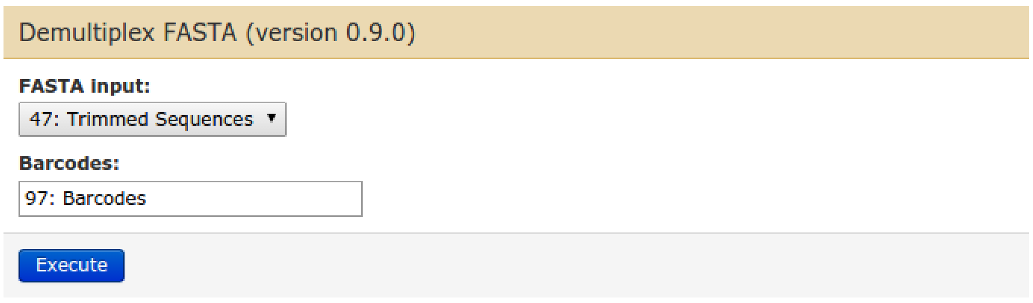
This tool extracts the FASTA track, the quality track and an xml summary from SFF binary files.

Figure 3.1 – Galaxy interface for the IS analysis pipeline (VISPA), with the center panel displaying the format conversion tool.

respectively, the sequences in FASTA format, the base qualities, an XML report and a log file. The following is the code of the XML wrapper:

```
<tool id="extract_sff" name="Extract SFF" version="0.9.0">
<command interpreter="python">
extract_sff.py '$sfffile' --logfile ${log_file}
-s '$fasta_track' -q '$qual_track' -x '$xml_track'
</command>
<inputs>
  <param name="sfffile" type="data" format="sff"
   label="454 SFF File"></param>
</inputs>
<outputs>
  <data name="fasta_track" format="fasta" label="Fasta Track"/>
  <data name="qual_track" format="qual" label="Qual Track"/>
  <data name="xml_track" format="xml" label="Xml report"/>
  <data name="log_file" format="txt" label="Log File"/>
</outputs>
</tool>
```

Note that the input dataset has its format set to “sff”. Besides being part of the metadata stored as a result of the tool being run, this information is used by the framework to filter out non-sff datasets from the drop-down menu, thus reducing the chance of errors. The tool wrapper also sets output


Demultiplex FASTA (version 0.9.0)

FASTA Input: 47: Trimmed Sequences

Barcodes: 97: Barcodes

Figure 3.2 – Galaxy interface for the demultiplex tool. Known barcodes are read from an additional input dataset.

dataset types, so that the same checks can be performed in the following steps.

Demultiplexing

This tool, implemented in Python, splits input sequences into several output files according to the leading barcode (sequence that do not contain any known barcode are discarded).

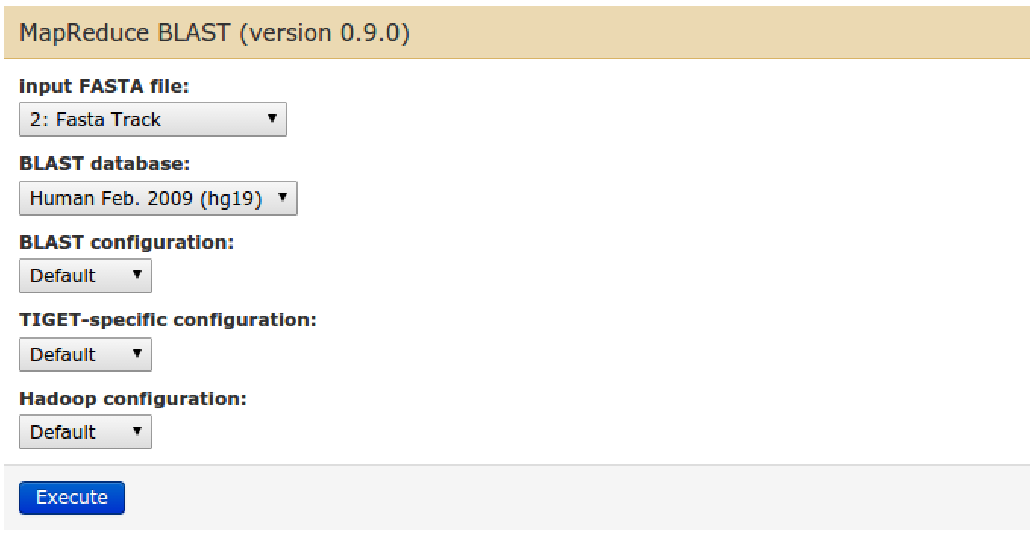
The list of known barcodes is given by an additional input dataset (see Fig. 3.2). Since in this case the number of output datasets cannot be known in advance, the tool writes a three-column HTML table where each row contains a barcode, the number of corresponding matches and a link to the actual output file.

Trimming

This tool reads a FASTA dataset containing the sequences to be trimmed. The LTR and LC are given as additional parameters in text boxes.

Alignment and Filtering

Fig.3.3 shows the Galaxy GUI for the distributed alignment and filtering step. Besides the list of input reads, the tool takes as input a BLAST database containing the reference genome. Due to their size, the time required to generate them and the fact that the possible choices are limited, these databases are stored in the Galaxy server (so that the user does not have to upload them) and made accessible via a drop-down menu. This is achieved in two steps: a tab-separated file where each row contains the option value (for the executable program), the name (for the GUI menu) and the path to the database is stored in the `tool-data` space of the Galaxy server:


MapReduce BLAST (version 0.9.0)

Input FASTA file: 2: Fasta Track

BLAST database: Human Feb. 2009 (hg19)

BLAST configuration: Default

TIGET-specific configuration: Default

Hadoop configuration: Default

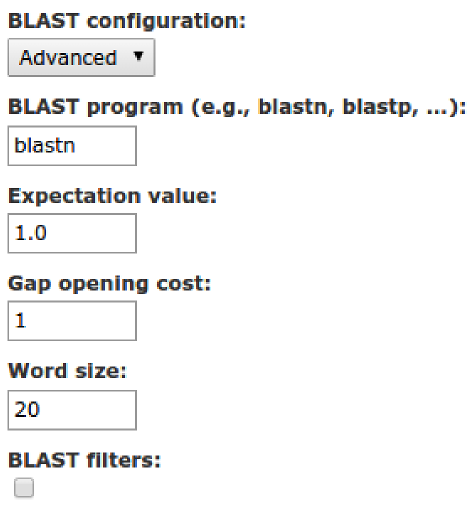
Figure 3.3 – Top-level Galaxy interface for the distributed alignment and filtering tool. Additional configuration sections appear if "advanced" is selected instead of "default" in the drop-down menus.

```
hg19  Human Feb. 2009 (hg19)  /path/to/hg19.tar
hg18  Human Mar. 2006 (hg18)  /path/to/hg18.tar
mm9   Mouse Jul. 2007 (mm9)   /path/to/mm9.tar
```

The drop-down menu is then automatically populated from the above file by adding the following code to the `inputs` section of the XML wrapper:

```xml
<param name="db" type="select" label="BLAST database">
  <options from_file="vispa_blastdb.loc">
    <column name="value" index="0"/>
    <column name="name" index="1"/>
    <column name="path" index="2"/>
  </options>
</param>
```

The Galaxy wrapper runs a Python driver script that launches the FASTA-to-tabular converter and the alignment and filtering Pydoop jobs in sequence, then collects classification results and writes the output files (see Sec. 2.2.2). The tool includes many configuration parameters, grouped into three categories: BLAST options, such as the expectation value and the word size; filtering options, such as the threshold values for the BLAST score and the homology score; Hadoop parameters (number of mappers for the two MapReduce jobs). To avoid cluttering the GUI, the wrapper uses a standard Galaxy


BLAST configuration:
Advanced

BLAST program (e.g., blastn, blastp, ...):
blastn

Expectation value:
1.0

Gap opening cost:
1

Word size:
20

BLAST filters:
☐

Figure 3.4 – Expansion of the BLAST options section in the alignment and filtering tool GUI.

mechanism where a configuration section is opened only if the user selects the "advanced" option from the corresponding drop-down menu (see Fig. 3.4). The following code fragment shows how this is implemented in the wrapper.

```
<conditional name="blast_conf">
  <param name="level" type="select" label="BLAST configuration">
    <option value="default" selected="true">Default</option>
    <option value="advanced">Advanced</option>
  </param>
  <when value="advanced">
    <param name="prog" size="10" type="text" value="blastn"
     label="BLAST program (e.g., blastn, blastp, ...)"/>
    <param name="evalue" size="10" type="float" value="1.0"
     label="Expectation value"/>
    <param name="gap_cost" size="10" type="integer" value="1"
     label="Gap opening cost"/>
    <param name="word_size" size="10" type="integer" value="20"
     label="Word size"/>
    <param name="filters" type="boolean" falsevalue=""
     truevalue="--blast-filters" label="BLAST filters"/>
  </when>
</conditional>
```

IS Merging

This step, implemented in Python, takes as input the unambiguous alignments list from the previous one and an optional window size for merging, and writes a tabular file containing the merged ISs.

IS Annotation

The annotation tool reads the merged ISs from the previous step and an annotation dataset in BED[3] format, and writes a tabular dataset containing the following information:

- The genomic position as a `(chromosome, offset)` pair, which identifies the IS;
- The feature's name and strand ($+/-$) as they appear in the annotation file;
- The feature's starting and ending offset;
- The distance between the IS and the feature's transcription start site (TSS);
- The relative position of the IS with respect to the feature (upstream, downstream or in-gene);
- The integration percentage, expressed as the ratio between the distance from the TSS and the total length of the feature.

3.2 Automated Interaction with Galaxy

Providing Galaxy wrappers for bioinformatics tools helps both accessibility and reproducibility: programs are easily manipulated under a common GUI, and execution histories keep track of datasets, computations and configuration parameters. In many cases, however, analysis tools must be accessed in a programmatic way, so that they can be used as components by external systems. For instance, consider a SNP discovery pipeline based on NGS sequence alignment. The results of this kind of analysis are never definitive: external events such as the release of a new reference genome, the availability of an improved version of the aligner, or the discovery of more accurate parameter settings are all capable of rendering said results outdated. In this

[3]http://genome.ucsc.edu/FAQ/FAQformat.html#format1

case, all existing output datasets must be recomputed from scratch. Another example of a situation where an analysis workflow must be re-executed programmatically is the fine-tuning of one or more parameters that have a statistical impact on the final results, after which the optimal value can be used for production runs.

Actions that can be described as — for instance — “repeat analysis workflow w three times, assigning the values v_1, v_2, v_3 to parameter k” are effectively *metacomputing* operations where iterations, compositions and so on are performed on programs or program sequences that act at the *computing* level, where the actual scientific analysis is performed. If metacomputing is performed on a regular basis, it requires the usage of specifically designed metacomputing engines. Otherwise, issues stemming from the lack of automation, some of which have been discussed in the introduction, are simply moved from the computing to the metacomputing level.

Building a metacomputing engine requires support from the computational frameworks being used, in the form of reliable ways to automate operations, perform bulk data processing and annotate analysis steps. However, having been designed for human interaction, graphical interfaces are hard to access programmatically. For this reason, Galaxy exposes its internals via a RESTful [52] API, allowing automated control by an external program. This interface is, however, fairly low-level: users must manually build and execute HTTP requests, explicitly handle error cases and perform JSON serialization and deserialization. This motivated the development of BioBlend [58], a Python library that takes care of HTTP communication, basic error handling and (de)serialization, allowing to interact with Galaxy entities such as histories, libraries and workflows via a dictionary-based interface.

Despite its significant improvements over the basic Galaxy API, the original BioBlend package missed several important features: the direct mapping of Python dictionaries to REST resources offered no explicit modeling of Galaxy entities and their relationships; the interface did not isolate client code from changes in the underlying Galaxy API; it lacked “rich” functionality capable of capturing higher-level abstractions, such as automatically retrieving all datasets for a given history or library.

This section describes BioBlend.objects, a Python object-oriented API for Galaxy interaction implemented as a higher-level layer above the original BioBlend. BioBlend.objects addresses the issues discussed above with two main features: an object-oriented programming model that simplifies development and decouples client code from changes in the underlying REST API; a high-level component that better supports metacomputing on Galaxy enti-

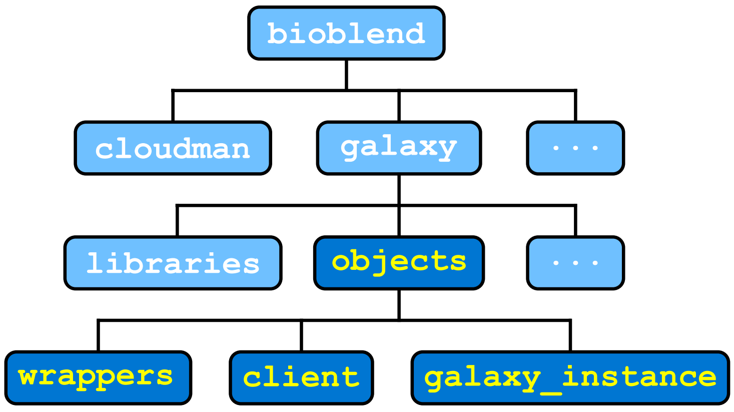


Figure 3.5 – BioBlend.objects modules as part of BioBlend's main structure. New modules are shown with their name in yellow on a darker background.

ties. BioBlend.objects has been included in the official BioBlend distribution[4] under `bioblend/galaxy/objects` (see Fig. 3.5).

3.2.1 Usage

The following code shows how to run a simple workflow that merges together the columns of two tabular files:

```
from bioblend.galaxy.objects import GalaxyInstance

gi = GalaxyInstance("URL", "API_KEY")
wf = gi.workflows.list()[0]
hist = gi.histories.list()[0]
inputs = hist.get_datasets()[:2]
input_map = dict(zip(wf.input_labels, inputs))
params = {"Paste1": {"delimiter": "U"}}
wf.run(input_map, "wf_output", params=params)
```

With the original BioBlend, the same task is accomplished with the following code:

```
from bioblend.galaxy import GalaxyInstance

gi = GalaxyInstance("URL", "API_KEY")
summaries = gi.workflows.get_workflows()
```

[4]https://github.com/galaxyproject/bioblend

```
wf_id = summaries[0]["id"]
wf_info = gi.workflows.show_workflow(wf_id)
hist_infos = gi.histories.get_histories()
hist_id = hist_infos[0]["id"]
hist_dict = gi.histories.show_history(hist_id)
content_info = gi.histories.show_history(hist_id, contents=True)
datasets = [gi.histories.show_dataset(hist_id, _["id"])
  for _ in content_info]
inputs = datasets[:2]
input_slots = wf_info["inputs"].keys()
input_map = {
  input_slots[0]: {"id": inputs[0]["id"], "src": "hda"},
  input_slots[1]: {"id": inputs[1]["id"], "src": "hda"}
  }
params = {"Paste1": {"delimiter": "U"}}
gi.workflows.run_workflow(wf_id, input_map,
  history_name="wf_output", params=params)
```

Even with this very simple example, where a workflow is retrieved from the Galaxy repository and executed after setting a parameter, the new API allows for much more compact code that is easier both to read and write.

3.2.2 Implementation

BioBlend.objects consists of three main components (see Fig. 3.5). The `wrappers` module defines the objects that represent Galaxy entities. All object classes derive from an abstract `Wrapper` that, conceptually (the actual code contains several additional optimizations and features), is structured as follows:

```
import abc

class Wrapper(object):
    __metaclass__ = abc.ABCMeta
    BASE_ATTRS = ("id", "name")

    @abc.abstractmethod
    def __init__(self, wrapped):
        self.wrapped = wrapped
        for k in self.BASE_ATTRS:
            setattr(self, k, self.wrapped.get(k))
```

Wrapper objects wrap the JSON-decoded dictionaries returned from the lower-level BioBlend interface, automatically defining attributes corresponding to each key. The `BASE_ATTRS` class attribute defines the "stable" interface, i.e., the set of attributes that are guaranteed to be available for each specific entity (`id` and `name` are common to all entities). This mechanism contributes to the isolation of client code from the dictionaries obtained from the Galaxy server.

The `client` module provides a high-level interface to a Galaxy instance based on the objects defined in `wrappers`. The module consists of three main classes that encapsulate interactions with the most important Galaxy entities: histories, workflows and libraries. These classes derive from a common `ObjClient` that, among other things, provides the core error handling functionality. Part of the code is shown in the following listing:

```
class ObjClient(object):

    def _error(self, msg, err_type=RuntimeError):
        self.log.error(msg)
        raise err_type(msg)

    def _get_dict(self, meth_name, reply):
        if reply is None:
            self._error("%s: no reply" % meth_name)
        elif isinstance(reply, collections.Mapping):
            return reply
        try:
            return reply[0]
        except (TypeError, IndexError):
            self._error("%s: unexpected reply: %r" %
                        (meth_name, reply))
```

Finally, the `galaxy_instance` module unifies the three clients into a `GalaxyInstance` object, which provides a single entry point for manipulating Galaxy entities:

```
import client

class GalaxyInstance(object):
    def __init__(self, url, api_key=None, email=None, password=None):
        self.gi = bioblend.galaxy.GalaxyInstance(
          url, api_key, email, password)
        self.histories = client.ObjHistoryClient(self)
        self.libraries = client.ObjLibraryClient(self)
        self.workflows = client.ObjWorkflowClient(self)
```

BioBlend.objects is used in OMERO.biobank (Sec. 1.4) as a workflow driver for the traceable (re)computation of genomics datasets (see Sec. 4.1).

Chapter 4

Enabling Traceability in Biomedical Applications

Sec. 1.4 introduced OMERO.biobank, a traceability framework for data-intensive biomedical research based upon a modified version of OMERO. The framework keeps track of provenance information by means of *actions* that link together the various entities. Actions are performed by *devices*, which represent transforming actors such as physical machines or software applications.

Fig. 4.1 shows a short example of entity-action chain: a flow cell [26] (which, in turn, contains biological samples, not shown in the diagram) is processed by a device consisting of an NGS machine (e.g., an Illumina HiSeq), which provides results in the form of a digital sequencer output. The latter is processed by a second device, this time a Galaxy (see Sec. 1.3) workflow (e.g., for format conversion and demultiplexing), generating one or more sequencing data samples.

In the graph database (see Fig.1.8), the `FlowCell`, `SequencerOutput` and `SeqDataSample` entities are represented by nodes, while actions are modeled as edges (shown as dashed arrows in Fig. 4.1). This mapping allows to quickly answer queries such as finding all sequencing data samples related to a given individual. The reason why such a query would be inefficient if performed directly on the backend is that in the object model the provenance chain flows in the opposite direction (solid arrows): each entity holds a reference to the action that produced it, which, in turn, refers to its target (the object it has been applied to). Since an action can lead to the production of an arbitrary number of objects, this convention allows to normalize the underlying one-to-many relationship.

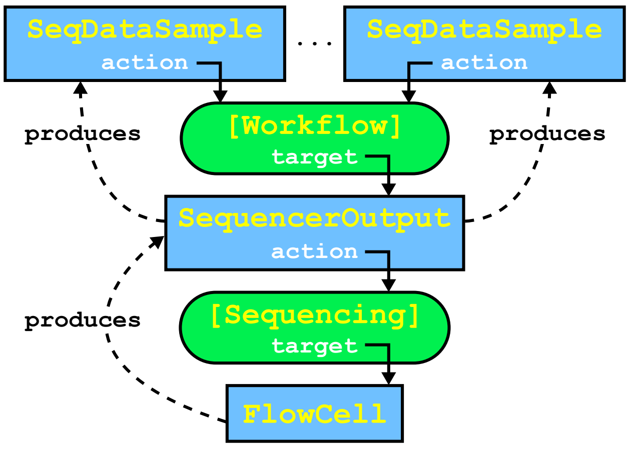


Figure 4.1 – An example of provenance chain in OMERO.biobank. Dashed arrows represent the direct mapping used in the graph DB, where an action is modeled as an edge between the producer and the product; solid arrows show how entities and actions are linked in the object model.

4.1 Integrating Galaxy and OMERO.biobank

OMERO.biobank's main goal is to support metacomputing at the provenance graph level. As discussed in Sec. 3.2, metacomputing refers to operations on entities which are themselves computing units, such as the iteration of a given analysis workflow with varying values of one or more parameters. In bioinformatics, in particular, three major classes of events, governed by different time scales, lead to the re-execution of whole analysis pipelines: the evolution of acquisition technology (e.g., the release of a new NGS machine), algorithmic improvements (such as the availability of a new read alignment software) and updates in the reference data (e.g., a new reference genome). To support metacomputing, the provenance information captured by the system must be machine-parsable, so that tasks like the recomputation of datasets produced by a complex pipeline can be automated. In OMERO.biobank, Galaxy workflows have been adopted as the standard mechanism for describing sequences of data transformations in a computable way. Although this is currently tied to a specific — albeit popular — execution engine, in the future it will be possible to describe workflows in a more portable way thanks to ongoing projects such as the Common Workflow Language (CWL)[1].

The `Workflow` entity derives from `SoftwareProgram`, in turn a specialization of `Device` (Fig. 4.2). The "body" attribute stores the JSON representation of the workflow as exported from Galaxy. This representation can be

[1]https://github.com/common-workflow-language/common-workflow-language

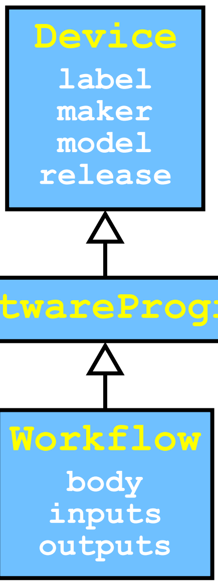


Figure 4.2 – The workflow entity in OMERO.biobank. Empty-headed arrows denote inheritance.

imported back into Galaxy (either in the same instance or in a distinct one), allowing the workflow to be re-executed. The "input" and "output" attributes store JSON-serialized annotations that provide the information required to map OMERO.biobank data collections to the input and output slots of the Galaxy workflow. A data collection (see Fig. 4.3) is a set of data samples, i.e., abstract representations of digital datasets. A data sample represents a dataset independently of how or where it is stored; such details are captured by data *objects* (derived from OMERO's `OriginalFile`), by means of attributes such as the path and the data type. Additionally, data samples can have different *roles* in the data collection. As discussed below, roles are used to correctly map datasets to input and output slots when driving external applications.

The following is an example of input annotation for a scaffolding workflow based upon SSPACE [8]:

```
{
  "contigs": {"name": "contigs", "mimetype": "x-vl/fasta"},
  "reads": {"name": "reads", "mimetype": "x-vl/fastq"},
  "mates": {"name": "mates", "mimetype": "x-vl/fastq"}
}
```

The top-level dictionary keys correspond to the role of each sample in the

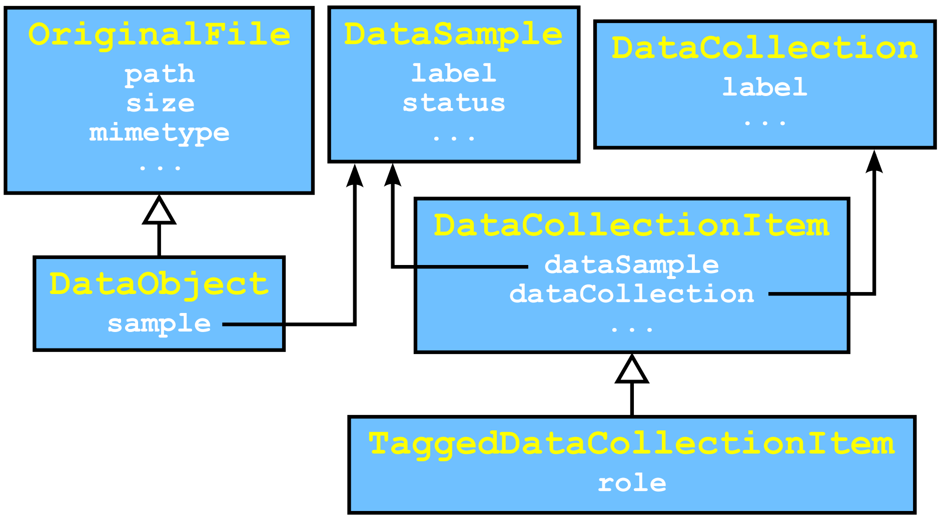


Figure 4.3 – DataCollection and related entities in OMERO.biobank. Since the relationship between data samples and collections is many-to-many, it is mediated by DataCollectionItem.

data collection. The `name` attribute contains the name of the corresponding slot in the Galaxy workflow, while `mimetype` specifies which of the collection's data objects can be mapped to the slot. Conversely, in the output annotation, type information allows to create output data objects that can be imported back into OMERO.biobank.

The following code fragment shows a simplified version of the workflow-driving section of the Galaxy adapter used in OMERO.biobank, implemented using BioBlend.objects (see Sec. 3.2).

```python
import json

class GalaxyHelper(object):

    def run_workflow(self, workflow, input_dc, **kwargs):
        input_annot = json.loads(workflow.inputs)
        galaxy_wf = self.gi.workflows.import_new(workflow.body)
        input_map = self.__build_input_map(input_annot, input_dc)
        out_hist_name = make_random_str()
        return galaxy_wf.run(input_map, out_hist_name, **kwargs)
```

The `run_workflow` method takes as input an OMERO.biobank workflow and a data collection where data sample roles must match keys in the workflow's input annotation. The body of the workflow is imported into

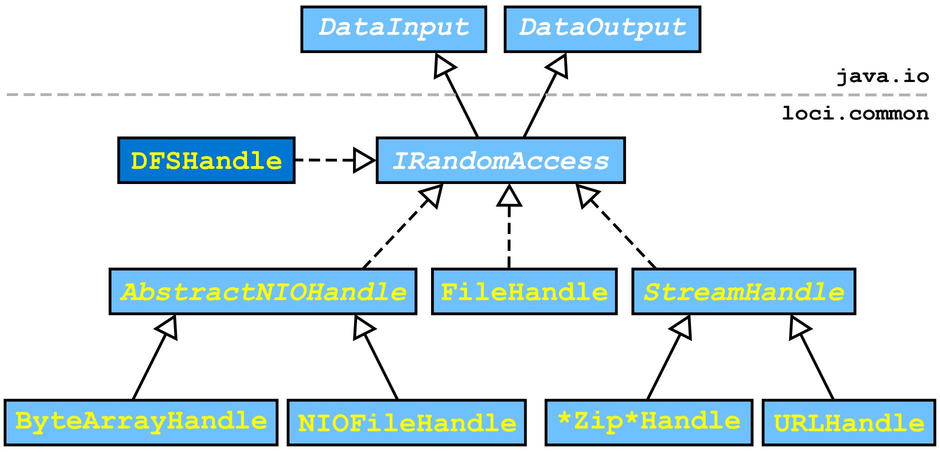


Figure 4.4 – `DFSHandle` location among Bio-Formats's I/O classes. Solid arrows denote inheritance, while dashed ones represent interface implementation.

Galaxy (`self.gi` is a reference to a BioBlend.objects `GalaxyInstance`), and run after mapping each data collection item to an input slot via the input annotation.

4.2 Integrating Hadoop and OMERO

In data-intensive projects, traceability and scalability are seldom standalone goals. Usually, applications must be able to efficiently scale with the input size and available resources while keeping track of all metadata required to reconstruct the provenance chain. While OMERO has been built right from the start with traceability as one of its main design goals, it currently lacks tight integration with high-performance distributed computing frameworks such as Hadoop (Sec. 1.2). This section describes preliminary work aimed at bridging the gap between the two technologies, allowing OMERO-based systems to use Hadoop as a server-side storage resource and computing engine.

As discussed in Sec. 1.4, OMERO uses Bio-Formats to access image data and metadata. Fig. 4.4 shows the main components of Bio-Formats's I/O structure: the root component, `IRandomAccess`, unifies Java's `DataInput` and `DataOutput` into a common interface for reading from and writing to a binary stream; in addition, it includes methods for setting and getting the current offset and byte order. Implementations of this interface include handles for regular files, byte arrays, URLs, etc.

The DFSHandle class[2] is a new implementation of IRandomAccess that wraps Hadoop's FSDataInputStream and FSDataOutputStream to provide an HDFS-aware backend for Bio-Formats. The following code fragment shows a simplified version of part of the Java implementation:

```java
public class DFSHandle implements IRandomAccess {
  /* ... */
  protected FSDataInputStream stream;
  protected FSDataOutputStream outStream;
  /* ... */
  public DFSHandle(String name, String mode, int bufferSize) {
    Configuration conf = new Configuration();
    fs = FileSystem.get(URI.create(name), conf);
    path = new Path(name);
    if (mode.equals("r")) {
      stream = (bufferSize < 0) ? fs.open(path) : fs.open(path,
          bufferSize);
    }
    else if (mode.equals("w")) {
      if (bufferSize < 0) {
        outStream = fs.create(path);
      }
      else {
        boolean overwrite = true;
        outStream = fs.create(path, overwrite, bufferSize);
      }
    }
    order = ByteOrder.BIG_ENDIAN;
  }
  /* ... */
  public long getFilePointer() {
    if (stream != null) return stream.getPos();
    else return outStream.getPos();
  }
  /* ... */
}
```

The DFSHandle object holds a reference to an HDFS input or output stream according to the opening mode. Most methods are implemented by

[2] https://github.com/simleo/bioformats/blob/hdfs/components/formats-common/src/loci/common/DFSHandle.java

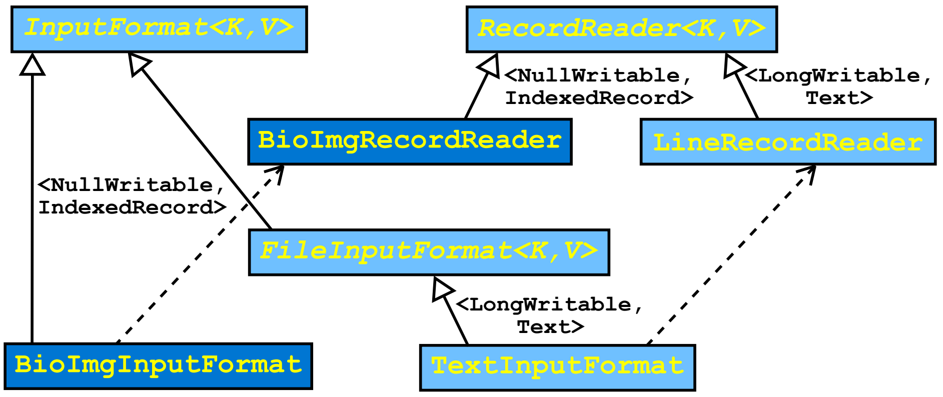


Figure 4.5 – Bio-Formats-specific classes (dark background) and their relationships with other input formats and record readers in Hadoop. Solid arrows denote extension, while dashed ones represent use (e.g., `LineRecordReader` is the type of the object returned by `TextInputFormat`'s `createRecordReader`)

delegating the required action to the wrapped stream, performing pre- and/or post-processing as required.

The new handle allows to read and process Bio-Formats-compatible data stored on HDFS; in particular, it enables OMERO import directly from Hadoop. Writing, however, is currently not possible in the general case, since `FSDataOutputStream` does not support the seek operation required by some formats.

Enabling HDFS read is only part of the work required to use image data as an input source for MapReduce applications. Hadoop's default input format, `TextInputFormat`, expects plain text files as input, and breaks them down into individual lines for consumption by the mapper (each text line is emitted as an input value, while the key is set to the line's byte offset in the file). Processing arbitrary binary files, however, requires the development of a custom input format. In practice, this is done by extending[3] `InputFormat`:

```
public abstract class InputFormat<K,V> {
  public abstract List<InputSplit> getSplits(JobContext context);
  public abstract RecordReader<K,V> createRecordReader(
      InputSplit split, TaskAttemptContext context);
}
```

The main task of an input format is to define how input data should be partitioned into subsets called *input splits*, each of which will be assigned

[3]In earlier versions of Hadoop, `InputFormat` was defined as an interface, so the required action was implementation rather than extension.

to a separate map task by the framework. This is done by implementing the `getSplits` method. The component that breaks down input splits into records is the *record reader*, which is also defined by subclassing a common abstract class:

```
public abstract class RecordReader<K,V> {
  public abstract void initialize(
      InputSplit split, TaskAttemptContext context);
  public abstract boolean nextKeyValue();
  public abstract K getCurrentKey();
  public abstract V getCurrentValue();
  public abstract void getProgress();
  public abstract void close();
}
```

The most important method in a record reader is `nextKeyValue`, which advances the reader's progress by one record. The method reads the next key and value and stores them so that they can be accessed by `getCurrentKey` and `getCurrentValue`. Since the record reader is usually tightly related to the input format, the latter provides the former via the `createRecordReader` method.

Fig. 4.5 shows how the new `BioImgInputFormat` relates to classes in the Hadoop MapReduce library. Most of Hadoop's built-in input formats, including `TextInputFormat`, derive from a common abstract `FileInputFormat` where input splits correspond to file chunks, identified by a path, a starting offset and a length (by default, each split corresponds to an HDFS block). `BioImgInputFormat`, in contrast, takes advantage of Bio-Formats's logical subdivision of image files into *series* (each series corresponds to a separate 5-dimensional pixel data structure) to create an input split for each series. `BioImgInputFormat` uses a specialized `BioImgRecordReader` that emits a serialized image plane as the value and no key (technically this is modeled as a `NullWritable` object that holds no data).

Hadoop provides a simple built-in serialization mechanism (formalized in the `Writable` interface) that covers commonly used data structures like primitive types, arrays, maps etc. However, this framework is not easily extended and, more importantly, not language-neutral. The latter property is particularly desirable in a scientific context, where analysis libraries are often written in languages other than Java. For this reason, in `BioImgInputFormat` data planes are serialized with Avro[4]. Avro data structures are expressed via

[4]https://avro.apache.org

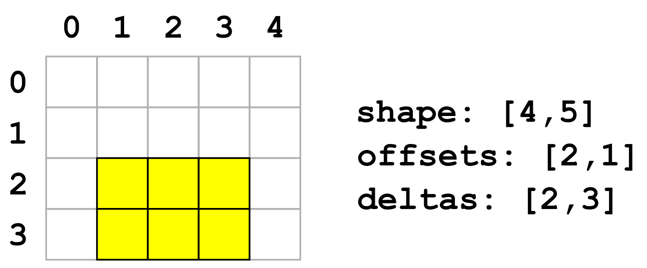


Figure 4.6 – A two-dimensional array slice (highlighted area).

schemas written in JSON, a data interchange format widely supported by modern programming languages. Avro is becoming increasingly popular in the bioinformatics community due to its adoption in the ADAM project [38]. Since authoring Avro schemas directly can be cumbersome, Avro supports a higher-level interface description language (IDL) that allows for more compact data structure definitions. The following listing shows the IDL descriptions used by `BioImgRecordReader`:

```
protocol DataBlock {
  enum DType {
    INT8, UINT8, INT16, UINT16, INT32, UINT32, FLOAT32, FLOAT64
  }
  record ArraySlice {
    DType dtype;
    boolean little_endian = true;
    array<int> shape;
    array<int> offsets;
    array<int> deltas;
    bytes data;
  }
}

protocol BioImg {
  record BioImgPlane {
    string name;
    string dimension_order;
    ArraySlice pixel_data;
  }
}
```

The `ArraySlice` structure models a generic, multidimensional array subsection. The `shape`, `offsets` and `deltas` attributes are monodimensional arrays with one element for each dimension in the array slice and contain, respectively, the total length, starting index and slice length for each dimension (a two-dimensional example is shown in Fig 4.6). The `data` attribute is a monodimensional byte array that stores actual data (the value corresponding to a given n-dimensional index can be determined based on the shape and data type `DType`). The `BioImg` structure is defined by a name (e.g., the file name with its extension stripped), a dimension order (e.g., `"XYZCT"`) and a data block of type `ArraySlice`. In practice, in the current version of OMERO and Bio-Formats, data blocks are always 5-dimensional, although this is expected to change in the future[5]. Full source code, including Avro definitions, is available at https://github.com/simleo/pydoop-features.

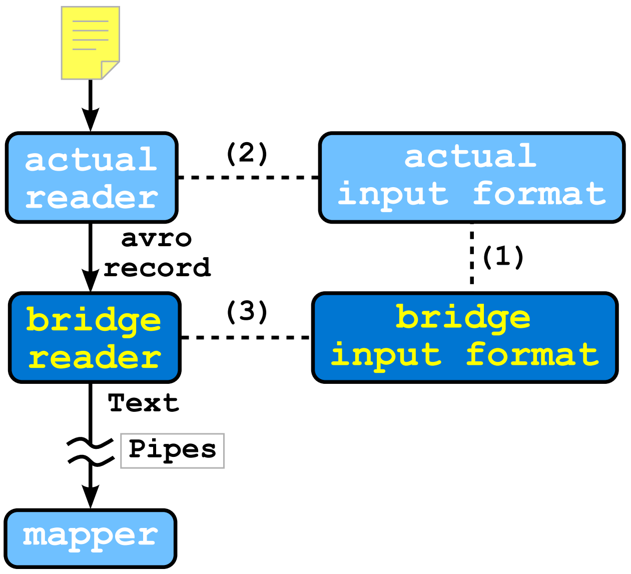


Figure 4.7 – Pydoop's Avro input bridge. The bridge input format reads the class name of the actual input format from the Hadoop configuration and creates an instance using reflection (1); the actual format is used to get the actual reader via `createRecordReader` (2); the actual reader is passed as a parameter to the bridge reader (3). The bridge reader repeatedly calls `nextKeyValue` on the actual reader, converts records and communicates them to the user's mapper over the Pipes protocol.

Due to its high popularity in the scientific community, interoperability with the Python language is particularly desirable. Although the Avro distribution includes, among others, a Python API, only the Java one provides support for MapReduce applications. The remainder of this section describes

[5] http://www.openmicroscopy.org/site/support/ome-model/developers/6d-7d-and-8d-storage.html

an interface that allows to use Avro for MapReduce development with Pydoop (see Sec. 1.2).

Pydoop enables MapReduce development by providing a Python client for the Hadoop Pipes protocol, which supports the implementation of mappers, reducers and other components in external, foreign-language user processes. Avro support has been added[6] to Pydoop via special *bridge* classes that perform on-the-fly conversion of Avro records to byte sequences that can be handled by the Pipes protocol.

Fig. 4.7 shows the logical structure of the Avro input bridge (the output case is similar, with roles reversed). Avro input mode is triggered when the user sets a specific parameter on the Pipes job submitter. In this case, the submitter configures the job to use the bridge input format instead of the actual one (which is expected to emit Avro records), whose class name is stored in a configuration parameter. The bridge input format reads the class name from the configuration and uses it to instantiate the actual format via Java reflection:

```java
public abstract class PydoopAvroInputBridgeBase<K, V>
    extends InputFormat<K, V> {

  protected InputFormat actualFormat;
  protected Class<? extends InputFormat> defaultActualFormat;

  protected InputFormat getActualFormat(Configuration conf) {
    if (actualFormat == null) {
      actualFormat = ReflectionUtils.newInstance(
          conf.getClass(
              Submitter.INPUT_FORMAT,
              defaultActualFormat,
              InputFormat.class), conf);
        }
    return actualFormat;
  }
}
```

The bridge calls `createRecordReader` on the actual input format to get the actual record reader which, in turn, is passed as a parameter to the bridge reader's constructor:

[6] https://github.com/crs4/pydoop/pull/98

```java
public class PydoopAvroInputValueBridge
    extends PydoopAvroInputBridgeBase<NullWritable,Text> {

  public PydoopAvroInputValueBridge() {
    defaultActualFormat = PydoopAvroValueInputFormat.class;
  }

  public RecordReader<NullWritable,Text> createRecordReader(
      InputSplit split, TaskAttemptContext context) {
    Configuration conf = context.getConfiguration();
    return new PydoopAvroBridgeValueReader(
        getActualFormat(conf).createRecordReader(split, context));
  }
}
```

Note that the above class is a specialized version of the base input bridge that expects Avro records to be exchanged over MapReduce values. Analogous subclasses handle the other two cases where records are exchanged, respectively, over keys and over both keys and values.

The bridge reader uses the actual reader passed to the constructor to fetch Avro records from the input source, converts them on the fly to byte arrays and wraps them in `Text` (a subtype of `Writable`) objects, which can be directly manipulated by the rest of the Pipes code and transmitted to the client. On the Python side, a specialized `AvroContext` object automatically deserializes records before passing them to the mapper, effectively hiding the entire serialization process from the user. Full source code is available at `http://crs4.github.io/pydoop`.

Chapter 5

Conclusion

Data-intensive biomedical studies have reached a level of complexity that makes them intractable with traditional, off-the-shelf computational tools. Although at the present time several specific technologies exist to tackle big data problems, combining them into a viable solution for the particular problem at hand is far from trivial, especially since the majority of projects require ad-hoc analysis, not covered by previously available software [15].

Scalability, or the ability to adapt to growing dataset sizes, has become an ubiquitous issue as more scientific fields are hit by the data deluge. Biomedical applications, due to the constantly increasing throughput capacity of wet-lab acquisition devices, are at the forefront of this revolution. Sec. 2.1 discussed a typical example of a readily available, state-of-the-art software tool that is perfectly adequate as long as the input size stays below a certain threshold (essentially dictated by the resources available on a single machine), but becomes very hard to use when that threshold is surpassed. The presented solution, in contrast, can take advantage of additional commodity hardware to automatically adapt to increasing dataset sizes, allowing to perform genotype calling at the genome-wide scale (millions of loci) on thousands of individuals at the same time. The alignment and filtering stage of the pipeline for viral integration site analysis, presented in Sec. 2.2, provides another example of scalable computational solution in a biomedical setting. Although in this case, as previously discussed, perfectly linear scalability is hard to achieve, the distributed implementation allowed to obtain an execution time compatible with the clinical studies the pipeline was designed for, enabling the discovery of novel therapeutic strategies for rare diseases [2, 7].

Reproducibility is another critical issue in computational biomedicine, as it enables experimental data to be shared and reused as the building blocks of new discoveries. Sec. 3.1 discussed the integration of the above bioinformatics pipeline into Galaxy, showing how the development of appropriate tools and wrappers can bring the computation inside the boundaries of a reproducibility-aware framework, providing the basic features required for the automatic replication of analysis workflows. The BioBlend.objects software library, presented in Sec. 3.2, builds upon such features to provide a compact, high-level programming interface that enables programmatic interaction with Galaxy resources, greatly simplifying the development of metacomputing routines.

Another highly desirable property of software systems for data-intensive science is traceability, i.e., the availability of provenance information for all entities stored in the system. To be actually usable in a programmatic way, thus enabling metacomputing, such information must be structured and *computable*, or machine-readable, as opposed to a simple disorganized collection of natural language notes. Sec. 4.1 introduced OMERO.biobank, a software framework for traceability management in biomedical applications, and discussed its integration with Galaxy. By modeling a workflow as a software device, and the analysis it performs as a transforming action that links the input and final output entities, provenance information can not only be retrieved, but also automatically interpreted to allow re-execution, parametrized iteration, composition and other metacomputing operations. To enable the inclusion of scalable components in such workflows, the traceability framework needs to provide an adequate level of integration with one or more distributed computing technologies. Sec. 4.2 described a Hadoop-aware I/O handler and input format for OMERO-based systems (such as OMERO.biobank), which together allow MapReduce applications to be scheduled directly from an OMERO backend, as well as an Avro adapter that allows to exchange complex, structured objects — such as those involved in most bioinformatics applications — between different programming languages in MapReduce jobs.

The work described here could be extended in several ways. To improve interoperability with other systems, call data produced by the genotype calling application could be written in one of the formats supported by ADAM (see Sec. 4.2). Additionally, Galaxy wrappers could be developed for the various steps involved in the process, allowing its integration as a workflow into OMERO.biobank. The workflow could then be used as a (re)computation engine for the traceable production of genotyping data samples from Affymetrix CEL datasets. Although the section of the Galaxy API covered by the current version of BioBlend.objects is sufficient for most applications, the object-oriented interface is not complete and could be strengthened by the inclusion of additional modules. Further investigation on the structure of biological image formats could lead to the development of a write-enabled Bio-Formats handle for HDFS files. More generally, any step towards a tighter integration of the various components would lead to a more robust, efficient and feature-rich framework for data-intensive biomedical analysis.

Bibliography

[1] David Abrahams and Ralf W. Grosse-Kunstleve. “Building hybrid systems with Boost.Python”. *C/C++ Users Journal* 21 (7), 2003, pp. 29–36.

[2] Alessandro Aiuti, Luca Biasco, Samantha Scaramuzza, et al. “Lentiviral Hematopoietic Stem Cell Gene Therapy in Patients with Wiskott-Aldrich Syndrome”. *Science* 341 (6148), 2013. DOI: 10.1126/science.1233151.

[3] Alessandro Aiuti, Shimon Slavin, Memet Aker, et al. “Correction of ADA-SCID by stem cell gene therapy combined with nonmyeloablative conditioning”. *Science* 296 (5577), 2002, pp. 2410–2413. DOI: 10.1126/science.1070104.

[4] Chris Allan, Jean-Marie Burel, Josh Moore, et al. “OMERO: flexible, model-driven data management for experimental biology”. *Nature Methods* 9 (3), 2012, pp. 245–253. DOI: 10.1038/nmeth.1896.

[5] Stephen F. Altschul, Warren Gish, Webb Miller, et al. “Basic local alignment search tool”. *Journal of Molecular Biology* 215 (3), 1990, pp. 403–410. DOI: 10.1016/S0022-2836(05)80360-2.

[6] Stephen D. Barr, Angela Ciuffi, Jeremy Leipzig, et al. “HIV integration site selection: targeting in macrophages and the effects of different routes of viral entry”. *Molecular Therapy* 14 (2), 2006, pp. 218–225. DOI: 10.1016/j.ymthe.2006.03.012.

[7] Alessandra Biffi, Eugenio Montini, Laura Lorioli, et al. “Lentiviral hematopoietic stem cell gene therapy benefits metachromatic leukodystrophy”. *Science* 341 (6148), 2013. DOI: 10.1126/science.1233158.

[8] Marten Boetzer, Christiaan V. Henkel, Hans J. Jansen, et al. “Scaffolding pre-assembled contigs using SSPACE”. *Bioinformatics* 27 (4), 2011, pp. 578–579. DOI: 10.1093/bioinformatics/btq683.

[9] B.M. Bolstad, R.A Irizarry, M. Åstrand, et al. "A comparison of normalization methods for high density oligonucleotide array data based on variance and bias". *Bioinformatics* 19 (2), 2003, pp. 185–193. DOI: 10.1093/bioinformatics/19.2.185.

[10] Kaan Boztug, Manfred Schmidt, Adrian Schwarzer, et al. "Stem-cell gene therapy for the Wiskott–Aldrich syndrome". *New England Journal of Medicine* 363 (20), 2010, pp. 1918–1927. DOI: 10.1056/NEJMoa1003548.

[11] Elizabeth Brauer, Dharmendra Singh, and Sorina Popescu. "Next-generation plant science: putting big data to work". *Genome Biology* 15 (1), 2014, p. 301. DOI: 10.1186/gb4149.

[12] Christian Jörg Braun, Kaan Boztug, Anna Paruzynski, et al. "Gene therapy for Wiskott-Aldrich syndrome — long-term efficacy and genotoxicity". *Science Translational Medicine* 6 (227), 2014, 227ra33. DOI: 10.1126/scitranslmed.3007280.

[13] William S. Bush and Jason H. Moore. "Chapter 11: Genome-Wide Association Studies". *PLoS Computational Biology* 8 (12), 2012, e1002822. DOI: 10.1371/journal.pcbi.1002822.

[14] Andrea Calabria, Simone Leo, Fabrizio Benedicenti, et al. "VISPA: a computational pipeline for the identification and analysis of genomic vector integration sites". *Genome Medicine* 6 (9), 2014. DOI: 10.1186/s13073-014-0067-5.

[15] Jeffrey Chang. "Core services: reward bioinformaticians". *Nature* 520, 7546 2015. DOI: 10.1038/520151a.

[16] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*. USENIX, 2004.

[17] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". *Communications of the ACM* 51 (1), 2008, pp. 107–113. DOI: 10.1145/1327452.1327492.

[18] Mike Folk, Gerd Heber, Quincey Koziol, et al. "An overview of the HDF5 technology suite and its applications". In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. 2011, pp. 36–47. DOI: 10.1145/1966895.1966900.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". *SIGOPS Operating Systems Review* 37 (5), 2003, pp. 29–43. DOI: 10.1145/1165389.945450.

[20] W. Wayt Gibbs. “Medicine gets up close and personal”. *Nature* 506 (7487), 2014, pp. 144–145. DOI: 10.1038/506144a.

[21] Carole Goble. “Better software, better research”. *IEEE Internet Computing* 18 (5), 2014, pp. 4–8. DOI: 10.1109/MIC.2014.88.

[22] Jeremy Goecks, Anton Nekrutenko, James Taylor, et al. “Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences”. *Genome Biology* 11 (8), 2010, R86. DOI: 10.1186/gb-2010-11-8-r86.

[23] Ilya Goldberg, Chris Allan, Jean-Marie Burel, et al. “The Open Microscopy Environment (OME) Data Model and XML file: open tools for informatics and quantitative analysis in biological imaging”. *Genome Biology* 6 (5), 2005, R47. DOI: 10.1186/gb-2005-6-5-r47.

[24] Salima Hacein-Bey-Abina, Alexandrine Garrigue, Gary P. Wang, et al. “Insertional oncogenesis in 4 patients after retrovirus-mediated gene therapy of SCID-X1”. *The Journal of Clinical Investigation* 118 (9), 2008, pp. 3132–3142. DOI: 10.1172/JCI35700.

[25] Anthony J. G. Hey, D. Stewart W. Tansley, and Kristin M. Tolle, eds. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, 2009.

[26] Robert A. Holt and Steven J.M. Jones. “The new paradigm of flow cell sequencing”. *Genome Research* 18 (6), 2008, pp. 839–846. DOI: 10.1101/gr.073262.107.

[27] S. Josefsson. *The base16, base32, and base64 data encodings*. RFC 4648. 2006. URL: http://www.rfc-editor.org/info/rfc4648.

[28] Joshua M. Korn, Finny G. Kuruvilla, Steven A. McCarroll, et al. “Integrated genotype calling and association analysis of SNPs, common copy number polymorphisms and rare CNVs”. *Nature Genetics* 40 (10), 2008, pp. 1253–1260. DOI: 10.1038/ng.237.

[29] Thomas LaFramboise. “Single nucleotide polymorphism arrays: a decade of biological, computational and technological advances”. *Nucleic Acids Research* 37 (13), 2009, pp. 4181–4193. DOI: 10.1093/nar/gkp552.

[30] Annalisa Lattanzi, Camilla Salvagno, Claudio Maderna, et al. “Therapeutic benefit of lentiviral-mediated neonatal intracerebral gene therapy in a mouse model of globoid cell leukodystrophy”. *Human Molecular Genetics* 23 (12), 2014, pp. 3250–3268. DOI: 10.1093/hmg/ddu034.

[31] Jeffrey T. Leek, Robert B. Scharpf, Héctor Corrada Bravo, et al. “Tackling the widespread and critical impact of batch effects in high-throughput data”. *Nature Reviews Genetics* 11 (10), 2010, pp. 733–739. DOI: 10.1038/nrg2825.

[32] Simone Leo, Luca Pireddu, Gianmauro Cuccuru, et al. “BioBlend.objects: metacomputing with Galaxy”. *Bioinformatics* 30 (19), 2014, pp. 2816–2817. DOI: 10.1093/bioinformatics/btu386.

[33] Simone Leo, Luca Pireddu, and Gianluigi Zanetti. “SNP genotype calling with MapReduce”. In: *Proceedings of the Third International Workshop on MapReduce and Its Applications*. 2012, pp. 49–56. DOI: 10.1145/2287016.2287026.

[34] Simone Leo and Gianluigi Zanetti. “Pydoop: a Python MapReduce and HDFS API for Hadoop”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 819–825. DOI: 10.1145/1851476.1851594.

[35] Melissa Linkert, Curtis T. Rueden, Chris Allan, et al. “Metadata matters: access to image data in the real world”. *The Journal of Cell Biology* 189 (5), 2010, pp. 777–782. DOI: 10.1083/jcb.201004104.

[36] Clifford Lynch. “Big data: How do your data grow?” *Nature* 455 (7209), 2008, pp. 28–29. DOI: 10.1038/455028a.

[37] Vivien Marx. “Biology: the big challenges of big data”. *Nature* 498, 7453 2013. DOI: 10.1038/498255a.

[38] Matt Massie, Frank Nothaft, Christopher Hartl, et al. *ADAM: genomics formats and processing patterns for cloud scale computing*. Tech. rep. UCB/EECS-2013-207. EECS Department, University of California, Berkeley, 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-207.html.

[39] Steven A. McCarroll, Finny G. Kuruvilla, Joshua M. Korn, et al. “Integrated detection and population-genetic analysis of SNPs and copy number variation”. *Nature Genetics* 40 (10), 2008, pp. 1166–1174. DOI: 10.1038/ng.238.

[40] William K. Michener, Suzie Allard, Amber Budden, et al. “Participatory design of DataONE—Enabling cyberinfrastructure for the biological and environmental sciences”. *Ecological Informatics* 11, 2012, pp. 5–15. DOI: 10.1016/j.ecoinf.2011.08.007.

[41] Greg Miller. “A scientist’s nightmare: software problem leads to five retractions”. *Science* 314, 5807 2006, p. 314.

[42] Paul R. Mueller and Barbara Wold. "In vivo footprinting of a muscle specific enhancer by ligation mediated PCR". *Science* 246 (4931), 1989, pp. 780–786. DOI: 10.1126/science.2814500.

[43] Raghunath Nambiar, Ruchie Bhardwaj, Adhiraaj Sethi, et al. "A look at challenges and opportunities of big data analytics in healthcare". In: *Proceedings of the 2013 IEEE International Conference on Big Data.* 2013, pp. 17–22. DOI: 10.1109/BigData.2013.6691753.

[44] Aisling O'Driscoll, Jurate Daugelaite, and Roy D. Sleator. "'Big data', Hadoop and cloud computing in genomics". *Journal of Biomedical Informatics* 46 (5), 2013, pp. 774–781. DOI: 10.1016/j.jbi.2013.07.001.

[45] Travis E. Oliphant. "Python for scientific computing". *Computing in Science Engineering* 9 (3), 2007, pp. 10–20. DOI: 10.1109/MCSE.2007.58.

[46] Elizabeth J. O'Neil. "Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM)". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data.* 2008, pp. 1351–1356. DOI: 10.1145/1376616.1376773.

[47] Marion G. Ott, Manfred Schmidt, Kerstin Schwarzwaelder, et al. "Correction of X-linked chronic granulomatous disease by gene therapy, augmented by insertional activation of MDS1-EVI1, PRDM16 or SETBP1". *Nature Medicine* 12 (4), 2006, pp. 401–409. DOI: 10.1038/nm1393.

[48] Chandra Shekhar Pareek, Rafal Smoczynski, and Andrzej Tretyn. "Sequencing technologies and genome sequencing". *Journal of applied genetics* 52 (4), 2011, pp. 413–35. DOI: 10.1007/s13353-011-0057-x.

[49] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. "SEAL: a distributed short read mapping and duplicate removal tool". *Bioinformatics* 27 (15), 2011, pp. 2159–2160. DOI: 10.1093/bioinformatics/btr325.

[50] Anna Pluzhnikov, Jennifer E. Below, Anuar Konkashbaev, et al. "Spoiling the whole bunch: quality control aimed at preserving the integrity of high-throughput genotyping". *American Journal of Human Genetics* 87 (1), 2010, pp. 123–128. DOI: 10.1016/j.ajhg.2010.06.005.

[51] Marco Ranzani, Stefano Annunziato, David J. Adams, et al. "Cancer gene discovery: exploiting insertional mutagenesis". *Molecular Cancer Research* 11 (10), 2013, pp. 1141–1158. DOI: 10.1158/1541-7786.MCR-13-0244.

[52] Leonard Richardson and Sam Ruby. *RESTful web services.* O'Reilly Media, 2007.

[53] Kate R. Rosenbloom, Joel Armstrong, Galt P. Barber, et al. "The UCSC Genome Browser database: 2015 update". *Nucleic Acids Research* 43 (D1), 2015, pp. D670–D681. DOI: 10.1093/nar/gku1177.

[54] Serena Sanna, Maristella Pitzalis, Magdalena Zoledziewska, et al. "Variants within the immunoregulatory CBLB gene are associated with multiple sclerosis". *Nature Genetics* 42 (6), 2010, pp. 495–497. DOI: 10.1038/ng.584.

[55] Manfred Schmidt, Kerstin Schwarzwaelder, Cynthia Bartholomae, et al. "High-resolution insertion-site analysis by linear amplification-mediated PCR (LAM-PCR)". *Nature Methods* 4 (12), 2007, pp. 1051–1057. DOI: 10.1038/nmeth1103.

[56] Terrence J. Sejnowski, Patricia S. Churchland, and J. Anthony Movshon. "Putting big data to good use in neuroscience". *Nature Neuroscience* 17 (11), 2014, pp. 1440–1441. DOI: 10.1038/nn.3839.

[57] Jay Shendure and Hanlee Ji. "Next-generation DNA sequencing". *Nature Biotechnology* 26 (10), 2008, pp. 1135–1145. DOI: 10.1038/nbt1486.

[58] Clare Sloggett, Nuwan Goonasekera, and Enis Afgan. "BioBlend: automating pipeline analyses within Galaxy and CloudMan". *Bioinformatics* 29 (13), 2013, pp. 1685–1686. DOI: 10.1093/bioinformatics/btt199.

[59] Thomas Sterling, Donald J. Becker, Daniel Savarese, et al. "Beowulf: a parallel workstation for scientific computation". In: *Proceedings of the 24th International Conference on Parallel Processing*. 1995, pp. 11–14.

[60] Herb Sutter. "The free lunch is over: a fundamental turn toward concurrency in software". *Dr. Dobb's journal* 30 (3), 2005, pp. 202–210.

[61] Ronald Taylor. "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics". *BMC Bioinformatics* 11 (Suppl 12), 2010, S1. DOI: 10.1186/1471-2105-11-S12-S1.

[62] Carol Tenopir, Suzie Allard, Kimberly Douglass, et al. "Data sharing by scientists: practices and perceptions". *PLoS ONE* 6 (6), 2011, e21101. DOI: 10.1371/journal.pone.0021101.

[63] Gudmundur A. Thorisson, Albert V. Smith, Lalitha Krishnan, et al. "The international HapMap project web site". *Genome Research* 15 (11), 2005, pp. 1592–1593. DOI: 10.1101/gr.4413105.

[64] Kristin M. Tolle, D. Stewart W. Tansley, and Anthony J. G. Hey. "The fourth paradigm: data-intensive scientific discovery". *Proceedings of the IEEE* 99 (8), 2011, pp. 1334–1337. DOI: 10.1109/JPROC.2011.2155130.

[65] Maria Valentini, Ilenia Zara, and Michele Muggiri. *Comparison of two strategies for genotype calling*. Poster presentation, ESHG 2011, Amsterdam, May 25–31. 2011.

[66] Ramona L. Walls, Balaji Athreya, Laurel Cooper, et al. "Ontologies as integrative tools for plant science". *American Journal of Botany* 99 (8), 2012, pp. 1263–1275. DOI: 10.3732/ajb.1200222.

[67] Gary P. Wang, Alexandrine Garrigue, Angela Ciuffi, et al. "DNA bar coding and pyrosequencing to analyze adverse events in therapeutic gene transfer". *Nucleic Acids Research* 36 (9), 2008, e49. DOI: 10.1093/nar/gkn125.

[68] Tom White. *Hadoop: the definitive guide*. 3rd ed. O'Reilly Media, 2012.

[69] Quan Zou, Xu-Bin Li, Wen-Rui Jiang, et al. "Survey of MapReduce frame operation in bioinformatics". *Briefings in Bioinformatics* 15 (4), 2014, pp. 637–647. DOI: 10.1093/bib/bbs088.