bistro Documentation

Release dev

Philippe Veber

May 01, 2020

Contents

CHAPTER 1

Getting started

This page describes how to install bistro, write a small pipeline and execute it.

1.1 Installation

bistro can be used under Linux and MacOSX (never tried with Windows). It can easily be installed using opam, the OCaml package manager. You can install opam following these instructions. Typically, under Debian/Ubuntu just

```
$ apt update && apt install opam
```

(as root or using sudo). Once this is done, initialize a fresh opam repository and install bistro:

```
$ opam init --comp=4.07.1
```

Take good care to follow the instructions given by opam after this command ends. Typically you will need to add this line:

```
$ . ${HOME}/.opam/opam-init/init.sh > /dev/null 2> /dev/null || true
```

to your .bashrc and execute

```
$ eval `opam config env`
```

for OPAM to be configured in your current console.

Now you're ready to install *bistro*, and utop which is a nice interactive interpreter for OCaml:

```
$ opam install bistro utop
```

If you're new to OCaml, you might want to install ocaml-top, which is a simple editor supporting syntax highlighting, automatic indentation and incremental compilation for OCaml:

```
$ opam install ocaml-top
```

You can also find similar support for other general-purpose editors like `emacs`, `vi` or `atom`.

1.2 A simple example

Using your favorite editor, create a file named `pipeline.ml` and paste the following program:

```
#require "bistro.bioinfo bistro.utils"

open Bistro
open Bistro_bioinfo
open Bistro_utils

let sample = Sra.fetch_srr "SRR217304"           (* Fetch a sample from the SRA␣
↪database *)
let sample_fq = Sra_toolkit.fastq_dump sample    (* Convert it to FASTQ format *)
let genome = Ucsc_gb.genome_sequence `sacCer2    (* Fetch a reference genome *)
let bowtie2_index = Bowtie2.bowtie2_build genome (* Build a Bowtie2 index from it *)
let sample_sam =                                 (* Map the reads on the reference␣
↪genome *)
  Bowtie2.bowtie2 bowtie2_index (`single_end [ sample_fq ])
let sample_peaks =                               (* Call peaks on mapped reads *)
  Macs2.(callpeak sam [ sample_sam ])

let repo = Repo.[
  [ "peaks" ] %> sample_peaks
]

(** Actually run the pipeline *)
let () = Repo.build_main ~outdir:"res" ~np:2 ~mem:(`GB 4) repo
```

1.3 Running a pipeline

A typical bioinformatics workflow will use various tools that should be installed on the system. Maintaining installations of many tools on a single system is particularly time-consuming and might become extremely tricky (e.g. to have several versions of the same tool, or tools that have incompatible dependencies on very basic pieces of the system, like the C compiler). To avoid this problem, `bistro` can use so-called *containers* like Docker or *Singularity <https://www.sylabs.io/>* to run each tool of the workflow in an isolated environment containing a proper installation of the tool. In practice, you don't have to install anything: for each step of a workflow `bistro` will invoke a container specifying which environment it needs. This is a tremendous time-saver in practice to deploy a pipeline on a new machine.

To get there you have to install `docker` or `singularity`. Follow instructions on this page for `` docker` and `this one <https://www.sylabs.io/guides/3.0/user-guide/quick_start.html#quick-installation-steps>`__ for ``singularity ``. Summarized instructions are also available there for `` docker`. Note that ``bistro `` can be used without containers, but in that case, you must make each program used in the pipeline available on your system.

Assuming `docker` is installed on your machine, you can simply run your pipeline by:

```
$ utop pipeline.ml
```

At the end you should obtain a `res` directory where you will find the output files of the pipeline.

In the remainder of this section we'll look at the code in more details, but first we'll need to learn a bit of the OCaml language.

CHAPTER 2

OCaml primer

Writing a workflow with `bistro` requires to learn a tiny subset of the OCaml language. This page aims at quickly presenting this subset, which should be sufficient to write basic pipelines. For the interested reader, I recommend the following easy introduction to the language and functional programming in general.

OCaml is a *functional language*, which means in brief that variables cannot be modified. The immediate consequence of this is that `for` or `while` loops are then pretty useless and are replaced by (possibly recursive) function calls. An OCaml program is a sequence of expressions (like `1 + 1`) or definitions introduced by the keyword `let`. For instance, the program

```
let a = 1 + 1
```

defines a variable named `a`, which has value `2`. This name can be reused in subsequent definitions, like in:

```
let a = 1 + 1
let b = 2 * a
```

A name cannot be used if it was not defined previously. If a name is used twice, the two definition coexist but only the last one is visible from the subsequent definitions. Hence, in the following program:

```
let a = 1 + 1
let b = 2 * a
let a = 1
let c = a
```

the variable `c` has value `1`.

2.1 Getting started with the OCaml interpreter

While OCaml programs can be compiled into executables, it is also very convenient to enter programs interactively using an *interpreter* (similar to what exists for `python` or `R`). The OCaml language has a very nice interpreter called utop than can easily installed using `opam`. In a shell just type:

```
opam install utop
```

and then you can call `utop` on the command line. An interpreter like `utop` reads expressions or definitions, evaluates them and prints the result. Expressions or definitions sent to `utop` should be ended with `;;` (in most cases they can be ommited in OCaml programs, but it doesn't hurt to keep them in the beginning). For instance, let's enter a simple sentence `let a = 1;;`. `utop` answers as follows:

```
     OCaml version 4.07.1

# let a = 1;;
val a : int = 1
```

The interpreter answers that we just defined a variable named `a`, of type `int` (the basic type for integers‘‘ and equal to `1`. Let's enter other definitions to meet new basic data types, like strings:

```
# let s = "bistro";;
val s : string = "bistro"
```

booleans:

```
# let b = true;;
val b : bool = true
```

or floating-point numbers:

```
# let x = 3.14159;;
val x : float = 3.14159
```

To quit the interpreter, just press `Ctrl+D`

2.2 Functions

In OCaml, functions can be defined with the `fun` keyword. For instance, the expression `fun x -> x + 1` denotes the function that given some integer returns the next integer. We can of course give the function a name like for any other value:

```
# let f = fun x -> x + 1;;
val f : int -> int = <fun>
```

Note that the interpreter "guessed" the type of `f`, as a function that takes an integer and returns an integer. This function can then be called using the following syntax:

```
# f 41;;
- : int = 42
```

In OCaml, the arguments of a function are just separated by spaces. In general we use a simpler (but equivalent) notation to define functions:

```
# let f x = x + 1;;
val f : int -> int = <fun>
```

Arguments can be named, in which case they are preceded by a ~ at the function definition and function calls:

```
# let f ~x = x + 1;;
val f : int -> int = <fun>
# f ~x:0;;
- : int = 1
```

Named arguments are very handy in that they can be given in any order; also they are a very effective way to document your code. A variant of named arguments are *optional arguments*, which may not be provided to the function.

Last, `bistro` API uses so-called *polymorphic variants*, which is a particular kind of values in OCaml. They are easy to spot because they are written with a leading backquote, like in:

```
# `mm10;;
- : [> `mm10 ] = `mm10
# `GB 3;;
- : [> `GB of int ] = `GB 3
```

The preceding snippet shows two basic usages of the variants: in the first one, they are used as a substitute to constant strings, the important difference being that the OCaml compiler will spot any typo at compile-time; the second usage is to wrap other values under a label that reminds of the meaning of the value. Here we define a memory requirement (3 GB), but instead of just representing it with an integer, we wrap it with the polymorphic variant to recall that this requirement is expressed in GB and not MB for instance.

CHAPTER 3

Basics: how to write pipelines

Now that we have a working installation of `bistro`, let us get back to our original goal, namely to write pipelines of scientific computations. In the following we'll use the `utop` interpreter to run an OCaml script. If you write your code in a file named `pipeline.ml`, you can run it by typing

```
$ utop pipeline.ml
```

which will create a `_bistro` directory used to store the results of the pipeline. We'll get back to that later, let's now start with an overview of the library.

3.1 What's in bistro

`bistro` essentially brings three main components:

- a data structure to represent a workflow, understood as a collection of interdependent steps
- an execution engine that can run a workflow, featuring parallel build, resume-on-failure and logging
- a library of pre-defined workflows to easily run applications from the field of computational biology

Those three components are provided as three libraries, respectively named `bistro`, `bistro.engine` and `bistro.bioinfo`. A fourth library named `bistro.utils` provides more convenient functions to run workflows and log execution.

One key feature of `bistro` is that workflows are described without ever caring for file names. The result of each computational step is automatically named and stored in a cache.

For a typical application, one will first describe the expected workflow either using already defined wrappers or by defining new ones. Once this is done, we define the outputs we want from the workflow, and how they should be layed out in an output directory (called an output repo). And finally we send this description to a build engine that will actually run the workflow.

3.2 A tiny QC pipeline

Let's write the above mentionned three parts on a simple example to perform quality check (QC) on a high-throughput sequencing sample. First, we need to load the library and open the appropriate modules:

```
#require "bistro.bioinfo bistro.utils"

open Bistro.EDSL
open Bistro_bioinfo.Std
open Bistro_utils
```

This will make the functions from the three components available. Then we can start writing our pipeline, with the following steps:

1. download a sample from the SRA database,
2. convert it to FASTQ format,
3. run FastQC on this data. Using the functions from the `bistro.bioinfo` library.

This is how it goes:

```
let sample = Sra.fetch_srr "SRR217304"
let sample_fq = Sra_toolkit.fastq_dump sample
let qc = FastQC.run sample_fq
```

Now we need to specify which output we are interested in, using the `Repo` module:

```
let repo = Repo.[
    ["qc"] %> qc ;
  ]
```

Here we specify that in our result directory, we want the output of `FastQC` to be named `qc`. The two other steps will not appear in the result directory, as we are not really interested in seeing them.

Finally, we can run the workflow using a function from the `Repo` module:

```
let () = Repo.build ~outdir:"res" repo
```

This will execute the workflow and place the result file we asked in it. You're now ready to actually run the pipeline: save the file and invoke

```
$ utop pipeline.ml
```

CHAPTER 4

How to wrap new tools

The library `bistro.bioinfo` offers a handful of functions to call various tools in computational biology, but of course many are missing. The purpose of this chapter is to demonstrate the few steps required to make a new tool available in `bistro` (a.k.a. wrapping).

4.1 A (very) basic example

As a starting example, let's see how we'd proceed with a very silly example, wrapping the `touch` command. To do so, we will use the `Bistro.Shell_dsl` module which provides many convenient functions to create new `workflow` values. Here's what it looks like:

```
open Bistro.Shell_dsl

let touch =
  Workflow.shell ~descr:"touch" [
    cmd "touch" [ dest ] ;
  ]
```

Let's describe what we wrote:

- the first line (open statement) makes all the many handy functions from `Bistro.Shell_dsl` visible in the current scope; many functions we describe below come from this module
- we define `touch` by calling a function from `Bistro.Workflow` named `shell`. As the name suggests, workflow steps it defines are built calling a command line on a shell.
- this function takes an argument `descr` which can be used to give a name to the workflow. This argument is optional and is only used for display purpose, but it helps `bistro` to display readable information when logging
- the second and last argument of `Workflow.shell` is a list of commands that will be executed when the workflow is run
- a command can be built with the `cmd` function from `Bistro.Shell_dsl`, which takes a string providing the name of the executable to run and a list of arguments

- arguments are of type `Bistro.Shell_dsl.template`, which can be seen as a representation of text with some special tokens inside, that can be replaced by some value when we try to execute the command
- the single argument to our command (`dest`) is an example of these special tokens, and represents a path where `bistro` expects to find the result file or directory of the workflow

Basically defining a workflow amounts to providing a list of commands that are expected to produce a result at the location represented by the token `dest`. **Note that a workflow that doesn't use ''dest'' is necessarily incorrect** since it has no means to produce its output at the expected location. The value `touch` we have defined has type `'a path workflow`, and represents a recipe (right, a very simple one) to produce a result file. This type is too general and we'd have to restrict it to prevent run-time error, but we'll see that later. Let's now see how we make make a pipeline on some parameter.

4.2 Parameterizing workflows

Our `touch` workflow is a very normal OCaml value. It's a datastructure that describes a recipe to produce a file. Let's write another one which is very similar:

```
let echo_hello =
  workflow ~descr:"echo_hello" [
    cmd "echo" ~stdout:dest [ string "hello" ] ;
  ]
```

There are a few newcomers here:

- there is an argument `stdout` to the `cmd` function, which adds to the command what's necessary to redirect its standard output to a file. Here we redirect to `dest`
- we see that we can form arguments from simple strings with the `string` function. There are other such argument constructors, like `int`, `float` and other more sophisticated ones

With this wrapper, we've encoded the following command line:

```
$ echo "hello" > $DEST
```

So far so good. But do we really have to write a new wrapper each time we want to change a small detail in the workflow? Of course not, instead we can simply write a function that produces our workflow:

```
let echo msg =
  workflow ~descr:"echo" [
    cmd "echo" ~stdout:dest [ string msg ] ;
  ]
```

Our workflow is now a lot more generic, since it can be used to produce files with any content. Well saying workflow here is slightly incorrect, because the value `echo` has type `string -> 'a path workflow`. It's a function that produces workflows, but since it will be so common, I'll just call them workflows. To put it another way, instead of writing a single script, we now have a function that can produce a particular kind of script given a string.

4.3 Depending on others

Most of the time, a computational step in a workflow will take as an input the results obtained from some other. This can be expressed thanks to the function `dep`. Let's see right away how it can be used to wrap the program `sort`:

```
let sort text_file =
  workflow ~descr:"sort" [
    cmd "sort" ~stdout:dest [ dep text_file ] ;
  ]
```

The value `sort` thus defined is again a function, but this time its argument is a workflow. If you ask OCaml, it will say that `sort` has type `'a path workflow -> 'b path workflow`. That is, given a first workflow, this function is able to build a new one. This new workflow will call `sort` redirecting the standard output to the expected destination and giving it `text_file` as an argument. More precisely, `bistro` will inject the location it decided for the output of workflow `text_file` in the command invocating `sort`. By combining the use of `dep` and `dest`, you can write entire collections of interdependent scripts without ever caring about where the generated files are stored.

4.4 Utility functions to describe a command's arguments

The functions `string` and `dep` are enough to describe virtually any command-line argument to a program. In addition, the module `Bistro.Shell_dsl` provides a few more utility functions that help writing concise and readable wrappers. The following code illustrates the use of a few of them on a simplified wrapper for the `bowtie` command:

```
let bowtie ?v index fq1 fq2 =
  workflow ~descr:"bowtie" [
    cmd "bowtie" [
      string "-S" ;
      opt "-1" dep fq1 ;
      opt "-2" dep fq2 ;
      option (opt "-v" int) v ;
      seq ~sep:"" [ dep index ; string "/index" ] ;
      dest ;
    ]
  ]
```

Let us examine each parameter to this command from top to bottom:

- the first argument is a simple `-S` switch, we encode it directly with the `string` function
- the second and third arguments are paths to input files introduces with a switch; here writing `[ ... ; opt "-1" dep fq1 ; ... ]` is equivalent to writing `[ ... ; string "-1" ; dep fq1 ; ... ]` but is shorter and more readable
- the fourth argument is optional; notice that the variable `v` is an optional argument to the `bowtie` function, so it is of type `'a option`; the `option` function from `Bistro.Shell_dsl` will add nothing to the command line if `v` is `None` or else apply its first argument to the value if holds. In that case, the applied function adds an integer argument introduced by a `-v` switch
- the fifth argument features a constructor called `seq` that can be used to concatenate a list of other chunks interspersed with a string (here the empty string); here we use it to describe a subdirectory of a workflow result
- the last argument is simply the destination where to build the result.

4.5 Typing workflows

We have seen that the `Workflow.shell` function from `Bistro.Shell_dsl` can be used to make new workflows that call external programs. This function has of course no means to know what the format of the result file or directory will be. For this reason, it outputs a value of type `'a path workflow`, which means a result whose format is compatible with any other. This is obviously wrong in the general case, and could lead to run-time errors by feeding a tool with inputs of an unsupported format. In order to prevent such run-time errors, we can provide more precise types to our functions producing workflows, when we have more information. Let's see that on an example. FASTA files have the property that when you concatenate several of them, the result is still a FASTA file (this is false in general case of course). We are now going to write a workflow that concatenates several FASTA files, and make sure its typing reflects this property.

Both `Bistro` and `Bistro_bioinfo` define a few type definitions for annotating workflows. In particular we'll use `Bistro_bioinfo.fasta` for our example. Here's how it looks:

```
open Bistro
open Bistro.Shell_dsl
open Bistro_bioinfo

let fasta_concat (x : fasta pworkflow) (y : fasta pworkflow) : fasta pworkflow =
  workflow ~descr:"fasta-concat" [
    cmd "cat" ~stdout:dest [ dep x ; dep y ] ;
  ]
```

Note the `'a pworkflow` type which is used here, and which is synonym for `'a path workflow`. Alternatively, you can define your workflow in a `.ml` file:

```
open Bistro.Shell_dsl

let fasta_concat x y =
  workflow ~descr:"fasta-concat" [
    cmd "cat" ~stdout:dest [ dep x ; dep y ] ;
  ]
```

and constraint its type in the corresponding `.mli` file:

```
open Bistro
open Bistro_bioinfo

val fasta_concat : fasta pworkflow -> fasta pworkflow -> fasta pworkflow
```

CHAPTER 5

Installing Docker

Docker is not absolutely required to use bistro but it brings quite some confort, since once you have it installed, you don't need to install the programs called by your pipeline. However the installation process on your machine can be a bit challenging, and while the documentation does provide all the necessary information to have a working docker installation, it can be a bit overwhelming to newcomers. The intent of this page is to provide a short recipe of how to install docker, hoping this recipe will work in most cases. Beware though, that the instructions may not be up to date or you may have a particular system configuration requiring adjustments, so this is in no way a substitute for the instructions given in docker documentation.

Contents

5.1 Debian

The full instructions are available there. Please go check this page (and drop me an email) if the instructions below don't work for you.

Perform the following commands as root user, or alternatively prefix all commands with sudo

```
$ apt update
$ apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg2 \
    software-properties-common
 $ curl -fsSL \
     https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg \
```

(continues on next page)

(continued from previous page)

```
    | apt-key add -
$ add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "
↪$ID") \
    $(lsb_release -cs) \
    stable"
$ apt-get update
$ apt-get install docker-ce
```

At this point `docker` should be installed, which you can check with the following command (still as root):

```
$ docker run hello-world
```

Now you need to make `docker` available for your normal user account. Let's say your login is `jdoe`, you need to execute:

```
$ usermod -aG docker jdoe
```

and quit your (graphical session) in order for this new configuration to be taken into account. Once your back, try as a normal user:

```
$ docker run hello-world
```

If this works, you're done!