BiOCamLib

BiOCamLib is the OCaml foundation upon which a number of the bioinformatics tools I developed are built.

It mostly consists of a library — you'll need to clone this repository if you want to manually compile other programs I've developed, notably SiNPle or KPop. You might also use the library for your own programs, if you are familiar with OCaml and patient enough to read the code.

As a bonus, BiOCamLib comes bundled with a few programs:

- `RC`, which can efficiently compute the reverse complement of (possibly very long) sequences. Each sequence should be input on a separate line — lines are processed one by one and not buffered. I use this program in many of my workflows.
- `Octopus`, which is a high-throughput program to compute the transitive closure of strings. This is useful to cluster things.
- `Parallel`, which allows you to split and process an input file chunk-wise using the reader/workers/writer model implemented in `BiOCamLib.Tools.Parallel`. You can see it as a demonstration of the capabilities of the library, but I also often use it as a useful tool to solve real-life problems, be they bioinformatics or not. A number of high-throughput real-life examples can be found in the KPop README.
- `FASTools`, which is a Swiss-knife tool for the manipulation of FASTA/FASTQ files. It supports all formats (FASTA, single- and paired-end FASTQ, interleaved FASTQ) and a simpler tabular format whereby FASTA/FASTQ records are represented as tab-separated lines. It facilitates format interconversions and other manipulations.

Installing `RC`, `Octopus`, `Parallel`, and `FASTools`

> ⚠️ Note that the only operating systems we officially support are Linux and MacOS. ⚠️
>
> Both OCaml and R are highly portable and you might be able to manually compile/install everything successfully on other platforms (for instance, Windows). Please let us know if you succeed or if you encounter some unexpected behaviour. However, please note that in general we are unable to provide installation-related support or troubleshooting on specific hardware/software combinations.

There are several possible ways of installing the software on your machine: through `conda`; by downloading pre-compiled binaries (Linux and MacOS x86_64 only); or manually.

Conda channel

BiOCamLib can be installed from the `bioconda` channel as package `biocamlib`. Just type

```
conda install -c bioconda biocamlib
```

or the equivalent command obtained replacing `conda` with `mamba` or `micromamba`, depending on which program you habitually use.

Pre-compiled binaries

You can download pre-compiled binaries for Linux and MacOS x86_64 from our releases. After doing so, just copy or move them to a directory which is accessible from your PATH.

For instance, supposing that you've downloaded programs to directory `~/.local/bin/`, in order to make them accessible from everywhere you'll have to execute a command such as

```
export PATH=~/.local/bin:$PATH
```

or add it to one of your login scripts (such as `~/.bashrc` or similar for `bash`).

Manual install

Alternatively, you can install `RC`, `Octopus`, `Parallel`, and `FASTools` manually by cloning and compiling the BiOCamLib sources. You'll need an up-to-date distribution of the OCaml compiler and the Dune package manager for that. Both can be installed through OPAM, the official OCaml distribution system. Once you have a working OPAM distribution you'll also have a working OCaml compiler, and Dune can be installed with the command

```
opam install dune
```

if it is not already present. Make sure that you install OCaml version 4.12 or later.

Then go to the directory into which you have downloaded the latest BiOCamLib sources, and type

```
./BUILD
```

That should generate the executables `RC`, `Octopus`, `Parallel`, and `FASTools`. Copy them to some favourite location in your PATH, for instance `~/.local/bin`.

Using `RC`

`RC` inputs sequences from standard input and outputs their reverse complement to standard output, one sequence at the time. Hence, `RC` can be conveniently used in a subprocess. For example, the command

```
echo GAtTaCA | RC
```

would produce `TGtAaTC`. Note that non-`[ACGTacgt]` characters are output unmodified, so sequence validation and linting must be performed elsewhere whenever they are necessary.

Command line options

This is the full list of command line options available for the program `RC`. You can visualise the list by typing

```
RC -h
```

in your terminal. You will see a header containing information about the version:

```
This is RC version 3 [02-Jan-2024]
 compiled against: BiOCamLib version 242 [23-Jan-2024]
 (c) 2023-2024 Paolo Ribeca <paolo.ribeca@gmail.com>
```

followed by detailed information. The general form(s) the command can be used is:

```
RC [OPTIONS]
```

Algorithm

Option	Argument(s)	Effect	Note(s)
`-C` `--no-complement`		do not base-complement the sequence	default=*base-complement*

Miscellaneous

Option	Argument(s)	Effect	Note(s)
`-V` `--version`		print version and exit	
`-h` `--help`		print syntax and exit	

Using `Octopus`

`Octopus` reads from its standard input equivalence relations, one set of relations per line. Each line consists of a set of strings separated by whitespace; if any two labels appear on the same line, they are considered to belong to the same equivalence class. When all the input has been parsed, `Octopus` outputs all the labels seen in the input sorted according to their equivalence class —

each line contains one equivalence class, with its member string labels separated by a `\t` character. The order in which classes appear is kept, but elements within the class will be lexicographically sorted. For example, the command

```
(cat <<___
A duh
  b  C
c f e
 duh zz x
b c
___
) | Octopus
```

will result in the output

```
A       duh     x       zz
C       b       c       e       f
```

(tab-separated).

Command line options for `Octopus`

This is the full list of command line options available for the program `Octopus`. You can visualise the list by typing

```
Octopus -h
```

in your terminal. You will see a header containing information about the version:

```
This is Octopus version 6 [02-Jan-2024]
 compiled against: BiOCamLib version 242 [23-Jan-2024]
 (c) 2016-2024 Paolo Ribeca <paolo.ribeca@gmail.com>
```

followed by detailed information. The general form(s) the command can be used is:

```
Octopus [OPTIONS]
```

Miscellaneous

Option	Argument(s)	Effect	Note(s)
`-V` `--version`		print version and exit	
`-h` `--help`		print syntax and exit	

Using `Parallel`

`Parallel` implements a subset of the functionality of GNU `parallel`, but is a compiled binary (hence potentially faster) and does not require you to install PERL or any other dependencies. It also has a much simpler interface. You can use it to transparently split a multi-line file into blocks having a specified size, which are then fed as standard input to a pool of worker threads. The number of threads is specified by the user, and the next block to be processed is assigned to the first thread that becomes available. The results of the processing of the blocks are concatenated in the order of the original blocks. If the number of threads is not specified, as many are used as the number of available processing units/CPU cores (as determined by the `nproc` command).

A typical bioinformatic use case might be something like

```
cat LargeFile.fasta | Parallel -l 1000 -- awk '{if ($0~"^>") print; else print toupper($0)}'
```

which would spawn as many workers as the number of available cores, split the input FASTA file into blocks of 1,000 lines each and process each block through `awk` to capitalise the sequence.

Keep in mind that there is little point in parallelising a job if sending/receiving data to/from it takes more time than the computation itself!

Command line options for `Parallel`

This is the full list of command line options available for the program `Parallel`. You can visualise the list by typing

```
Parallel -h
```

in your terminal. You will see a header containing information about the version:

```
This is Parallel version 8 [18-Jan-2024]
 compiled against: BiOCamLib version 242 [23-Jan-2024]
 (c) 2019-2024 Paolo Ribeca <paolo.ribeca@gmail.com>
```

followed by detailed information. The general form(s) the command can be used is:

```
Parallel [OPTIONS] -- [COMMAND TO PARALLELIZE AND ITS OPTIONS]
```

Command to parallelize

Option	Argument(s)	Effect	Note(s)
`--`		consider all the subsequent parameters as the command to be executed in parallel. At least one command must be specified	*(mandatory)*

Input/Output

Option	Argument(s)	Effect	Note(s)
`-l` `--lines-per-block`	*positive_integer*	number of lines to be processed per block	default=*10000*
`-i` `--input`	*input_file*	name of input file	default=*stdin*
`-o` `--output`	*output_file*	name of output file	default=*stdout*

Miscellaneous

Option	Argument(s)	Effect	Note(s)
`-t` `--threads`	*positive_integer*	number of concurrent computing threads to be spawned (default automatically detected from your configuration)	default=*nproc*
`-v` `--verbose`		set verbose execution	default=*false*
`-d` `--debug`		output debugging information	default=*false*
`-V` `--version`		print version and exit	
`-h` `--help`		print syntax and exit	

Using `FASTools`

`FASTools` allows you to manipulate FASTA/FASTQ files by offering a rich set of tools to convert FASTA/FASTQ records to/from a tab-separated format. Together with other programs such as `awk`, that allows a large set of operations to be implemented effortlessly.

For example:

```
cat Sequences.fasta | FASTools | awk -F '\t' '{print ">"$1"\n"substr($2,1,int(length($2)+1)/2)}'
```

puts each FASTA record in file `Sequences.fasta` on a single line, with the sequence name and the sequence being separated by a `'\t'` character. The `awk` part then reads each line, replaces the sequence with its first half, and re-outputs the sequence as a FASTA record.

Due to default command line option values, the command is actually equivalent to

```
cat Sequences.fasta | FASTools c -F | awk -F '\t' '{print ">"$1"\n"substr($2,1,int(length($2)+1)/2)}'
```

and, if one uses forms accepting a direct rather than a piped input, to

```
FASTools -f Sequences.fasta | awk -F '\t' '{print ">"$1"\n"substr($2,1,int(length($2)+1)/2)}'
```

or to

```
FASTools c -f Sequences.fasta | awk -F '\t' '{print ">"$1"\n"substr($2,1,int(length($2)+1)/2)}'
```

Similar command line options are provided to manipulate as single lines FASTQ and tabular formats rather than FASTA. In detail:

- Option `-f <FASTA_file>` operates on a FASTA file given as an explicit argument, while `-F` processes the same kind of file piped into the program as standard input
- Option `-s <FASTQ_file>` operates on a FASTQ file containing single-end reads given as an explicit argument, while `-S` processes the same kind of file piped into the program as standard input
- Option `-p <FASTQ_file_1> <FASTQ_file_2>` operates on two FASTQ file separately containing first and second ends of paired-end reads given as an explicit argument, while `-P` processes an interleaved FASTQ file containing alternating first and second ends of paired-end reads piped into the program as standard input
- Option `-t <tabular_file>` operates on a tabular file containing FASTA/FASTQ records compacted on a single line by FASTools given as an explicit argument, while `-T` processes the same kind of file piped into the program as standard input. Note that there are exactly three possible tabular formats recognised by FASTools:
  i. Lines of the form `<read_name> '\t' <read_sequence>`, corresponding to a compacted FASTA record
  ii. Lines of the form `<read_name> '\t' <read_sequence> '\t' <read_qualities>`, corresponding to a compacted single-end FASTQ record
  iii. Lines of the form `<read_name_1> '\t' <read_sequence_1> '\t' <read_qualities_1> '\t' <read_name_2> '\t' <read_sequence_2> '\t' <read_qualities_2>`, corresponding to a compacted paired-end FASTQ record.

Several command switches exist that natively perform operations on sequence/qualities within `FASTools`, with no need to write any additional external code:

- Option `c` or `-c`, a shorthand for `compact`, will group each FASTA/FASTQ record on a single tab-separated line. It is the default
- Option `e` or `-e`, a shorthand for `expand`, will split tabular records into multi-line FASTA/FASTQ records. It is the inverse of `c`. Note that it is perfectly fine to apply option `e` to an existing input which is already in FASTA/FASTQ format, in which case `FASTools` will normalise the input by putting each sequence on a single line, which is how `FASTools` outputs sequences. Linting filters (option `-l`) can also be used to further normalise the sequence
- Option `m <regexp>` or `-m <regexp>`, a shorthand for `match <regexp>`, will select FASTA/FASTQ records whose name match ''. Note that `<regexp>` must be defined according to https://ocaml.org/api/Str.html
- Option `r` or `-r`, a shorthand for `revcom`, will reverse-complement the sequence, and reverse the qualities if present, of FASTA/FASTQ records
- Option `d` or `-d`, a shorthand for `dropq`, will drop qualities from FASTA/FASTQ records, turning a FASTQ into a FASTA file and having no effect on a FASTA file.

Repeated command line options can be seen as different commands that are executed in order of specification. For instance,

```
FASTools m "^CONTIG_1$" -f Assembly_1.fasta m "^CONTIG_42$" -f Assembly_2.fasta
```

will select sequence `CONTIG_1` from file `Assembly_1.fasta` and sequence `CONTIG_42` from file `Assembly_2.fasta`, outputting one after the other. Note however that each `c` / `e` / `m` / `r` / `d` option will input and output precisely one file, so option `c` in

```
FASTools e -f Multiline.fasta c
```

will not have any effect — the result will be a FASTA file with sequences on the same line, not a tabular one. To perform on the same input more than one transformation requiring I/O, you'll have to pipe multiple `FASTools` commands one after the other, as in

```
FASTools e -f Multiline.fasta | FASTools c
```

which will give the expected result.

A few more examples:

- ```
  FASTools e -p <(zcat Sample76_1.fq.gz) <(zcat Sample76_2.fq.gz)
  ```

  will interleave compressed files, while
- ```
  FASTools -s Reads.fastq | shuf | awk '((NR-1)%10==0)' | FASTools e -T
  ```

  will randomly select one read in 10. Note that the corresponding version with paired-end files in input, namely:
- ```
  FASTools -p Reads_1.fastq Reads_2.fastq | shuf | awk '((NR-1)%10==0)' | FASTools e -T
  ```

  will still work correctly and produce an interleaved output.

Finally, you can combine `FASTools` with `Parallel` to distribute computation across different CPUs. For instance, the command:

```
FASTools -f Sequences.fasta | Parallel -- awk -F '\t' '{print ">"$1"\n"substr($2,1,int(length($2)+1)/2)}'
```

will still cut sequences in half as the first example, but it will do so by splitting the input file compacted by FASTools into blocks of 10,000 records each and by distributing the computation among all the available CPU cores.

Command line options for `FASTools`

This is the full list of command line options available for the program `FASTools`. You can visualise the list by typing

```
FASTools -h
```

in your terminal. You will see a header containing information about the version:

```
This is FASTools version 8 [18-Mar-2024]
 compiled against: BiOCamLib version 245 [14-Feb-2024]
 (c) 2022-2024 Paolo Ribeca <paolo.ribeca@gmail.com>
```

followed by detailed information. The general form(s) the command can be used is:

```
FASTools [OPTIONS]
```

Working mode. Executed delayed in order of specification, default=*compact*.
```

Option	Argument(s)	Effect	Note(s)
`compact` `-c` `--compact`		put each FASTA/FASTQ record on one tab-separated line	
`expand` `-e` `--expand`		split each tab-separated line into one or more FASTA/FASTQ records	
`match` `-m` `--match`	*regexp*	select sequence names matching the specified regexp in FASTA/FASTQ records or tab-separated lines. The regexp must be defined according to https://ocaml.org/api/Str.html. For paired-end files, the pair matches when at least one name matches	
`revcom` `-r` `--revcom`		reverse-complement sequences (and reverse qualities if present) in FASTA/FASTQ records or tab-separated lines	
`dropq` `-d` `--dropq`		drop qualities in FASTA/FASTQ records or tab-separated lines	

Input/Output. Executed delayed in order of specification, default=*-F*.

Option	Argument(s)	Effect	Note(s)
`-f` `--fasta`	*fasta_file_name*	process FASTA input file containing sequences	
`-F`		process FASTA sequences from standard input	
`-s` `--single-end`	*fastq_file_name*	process FASTQ input file containing single-end sequencing reads	
`-S`		process single-end FASTQ sequencing reads from standard input	
`-p` `--paired-end`	*fastq_file_name1* *fastq_file_name2*	process FASTQ input files containing paired-end sequencing reads	
`-P`		process interleaved FASTQ sequencing reads from standard input	
`-t` `--tabular`	*tabular_file_name*	process input file containing FAST[A|Q] records as tab-separated lines	
`-T`		process FAST[A|Q] records in tabular form from standard input	
`-l` `--linter`	`none` | `DNA` | `dna` | `protein`	sets linter for sequence. All non-base (for DNA) or non-AA (for protein) characters are converted to unknowns	default=*none*
`--linter-keep-lowercase`	`true` | `false`	sets whether the linter should keep lowercase DNA/protein characters appearing in sequences rather than capitalise them	default=*false*
`--linter-keep-dashes`	`true` | `false`	sets whether the linter should keep dashes appearing in sequences rather than convert them to unknowns	default=*false*
`-o` `--output`	*output_file_name*	set the name of the output file. Files are kept open, and it is possible to switch between them by repeatedly using this option. Use `/dev/stdout` for standard output	default=*/dev/stdout*
`-O` `--paired-end-output`	*output_file_name_1* *output_file_name_2*	set the names of paired-end FASTQ output files. Files are kept open, and it is possible to switch between them by repeatedly using this option. Use `/dev/stdout` for standard output	default=*/dev/stdout*
`--flush` `--flush-output`		flush output after each record (global option)	default=*do not flush*

Miscellaneous

Option	Argument(s)	Effect	Note(s)
`-v` `--verbose`		set verbose execution (global option)	default=*false*
`-V` `--version`		print version and exit	
`-h` `--help`		print syntax and exit	