Sailfish Documentation

Release 0.7.6

Rob Patro, Carl Kingsford and Steve Mount

Apr 24, 2017

Contents

Contents:

CHAPTER 1

Requirements

- A C++11 conformant compiler (currently tested with GCC>=4.7 and Clang>=3.4)
- CMake. Sailfish uses the CMake build system to check, fetch and install dependencies, and to compile and install Sailfish. CMake is available for all major platforms (though Sailfish is currently unsupported on Windows.)

CHAPTER 2

Installation

After downloading the Sailfish source distribution and unpacking it, change into the top-level directory:

```
> cd Sailfish
```

Then, create and out-of-source build directory and change into it:

```
> mkdir build
> cd build
```

Sailfish makes extensive use of Boost. We recommend installing the most recent version (1.55) systemwide if possible. If Boost is not installed on your system, the build process will fetch, compile and install it locally. However, if you already have a recent version of Boost available on your system, it make sense to tell the build system to use that.

If you have Boost installed you can tell CMake where to look for it. Likewise, if you already have Intel's Threading Building Blocks library installed, you can tell CMake where it is as well. The flags for CMake are as follows:

- -DFETCH_BOOST=TRUE – If you don't have Boost installed (or have an older version of it), you can provide the FETCH_BOOST flag instead of the BOOST_ROOT variable, which will cause CMake to fetch and build Boost locally.
- -DBOOST_ROOT=<boostdir> – Tells CMake where an existing installtion of Boost resides, and looks for the appropritate version in <boostdir>. This is the top-level directory where Boost is installed (e.g. /opt/local).
- -DTBB_INSTALL_DIR=<tbbroot> – Tells CMake where an existing installation of Intel's TBB is installed (<tbbroot>), and looks for the apropriate headers and libraries there. This is the top-level directory where TBB is installed (e.g. /opt/local).
- -DCMAKE_INSTALL_PREFIX=<install_dir> – <install_dir> is the directory to which you wish Sailfish to be installed. If you don't specify this option, it will be installed locally in the top-level directory (i.e. the directory directly above "build").

Setting the appropriate flags, you can then run the CMake configure step as follows:

```
> cmake [FLAGS] ..
```

The above command is the cmake configuration step, which *should* complain if anything goes wrong. Next, you have to run the build step. Depending on what libraries need to be fetched and installed, this could take a while (specifically if the installation needs to install Boost). To start the build, just run make.

```
> make
```

If the build is successful, the appropriate executables and libraries should be created. There are two points to note about the build process. First, if the build system is downloading and compiling boost, you may see a large number of warnings during compilation; these are normal. Second, note that CMake has colored output by default, and the steps which create or link libraries are printed in red. This is the color chosen by CMake for linking messages, and does not denote an error in the build process.

Finally, after everything is built, the libraries and executable can be installed with:

```
> make install
```

To ensure that Sailfish has access to the appropriate libraries you should ensure that the PATH variabile contains <install_dir>/bin, and that LD_LIBRARY_PATH (or DYLD_FALLBACK_LIBRARY_PATH on OSX) contains <install_dir>/lib.

After the paths are set, you can test the installation by running

```
> make test
```

This should run a simple test and tell you if it succeeded or not.

CHAPTER 3

Sailfish

Sailfish is a tool for transcript quantification from RNA-seq data. It requires a set of target transcripts (either from a reference or *de-novo* assembly) to quantify. All you need to run sailfish is a fasta file containing your reference transcripts and a (set of) fasta/fastq file(s) containing your reads. Sailfish runs in two phases; indexing and quantification. The indexing step is independent of the reads, and only needs to be run once for a particular set of reference transcripts and choice of k (the k-mer size). The quantification step, obviously, is specific to the set of RNA-seq reads and is thus run more frequently.

Indexing

To generate the sailfish index for your reference set of transcripts, you should run the following command:

```
> sailfish index -t <ref_transcripts> -o <out_dir> -k <kmer_len>
```

This will build a sailfish index using k-mers of length `<kmer_len>` for the reference transcripts provided in the file `<ref_transcripts>` and place the index under the directory `<out_dir>`. There are additional options that can be passed to the sailfish indexer (e.g. the number of threads to use). These can be seen by executing the command `sailfish index -h`.

Note that, as of v0.7.0, the meaning of the `-k` parameter has changed slightly. Rather than the k-mer size on which Sailfish will quantify abundances, it becomes the minimum match size that will be considered in the quasi-mapping procedure during quantification. For sufficiently long (e.g. 75bp or greater) reads, the default should be acceptable. If you have substantially shorter reads, you may want to consider a smaller `-k`.

Note: values of k

The `k` value used to build the Sailfish index must be an odd number. Using an even value for `k` will raise an error and the full index will not be built.

Quantification

Now that you have generated the sailfish index (say that it's the directory <index_dir> — this corresponds to the <out_dir> argument provided in the previous step), you can quantify the transcript expression for a given set of reads. To perform the quantification, you run a command like the following:

```
> sailfish quant -i <index_dir> -l "<libtype>" {-r <unmated> | -1 <mates1> -2 <mates2>
↪} -o <quant_dir>
```

Where <index_dir> is, as described above, the location of the sailfish index, <libtype> is a string describing the format of the fragment (read) library (see *Fragment Library Types*), <unmated> is a list of files containing unmated reads, <mates{1,2}> are lists of files containg, respectively, the first and second mates of paired-end reads. Finally, <quant_dir> is the directory where the output should be written. Just like the indexing step, additional options are available, and can be viewed by running `sailfish quant -h`.

When the quantification step is finished, the directory <quant_dir> will contain a file named "quant.sf" (and, if bias correction is enabled, an additional file names "quant_bias_corrected.sf"). This file contains the result of the Sailfish quantification step. This file contains a number of columns (which are listed in the last of the header lines beginning with '#'). Specifically, the columns are (1) Transcript ID, (2) Transcript Length, (3) Transcripts per Million (TPM) and (6) Estimated number of reads (an estimate of the number of reads drawn from this transcript given the transcript's relative abundance and length). The first two columns are self-explanatory, the next four are measures of transcript abundance and the final is a commonly used input for differential expression tools. The Transcripts per Million quantification number is computed as described in[1], and is meant as an estimate of the number of transcripts, per million observed transcripts, originating from each isoform. Its benefit over the F/RPKM measure is that it is independent of the mean expressed transcript length (i.e. if the mean expressed transcript length varies between samples, for example, this alone can affect differential analysis based on the K/RPKM.).

Description of important options

Sailfish exposes a number of useful optional command-line parameters to the user. The particularly important ones are explained here, but you can always run `sailfish quant -h` to see them all.

-p / --numThreads

The number of threads that will be used for quasi-mapping, quantification, and bootstrapping / posterior sampling (if enabled). Sailfish is designed to work well with many threads, so, if you have a sufficient number of processors, larger values here can speed up the run substantially.

--useVBOpt

Use the variational Bayesian EM algorithm rather than the "standard" EM algorithm to optimize abundance estimates. The details of the VBEM algorithm can be found in[2], and the details of the variant over fragment equivalence classes that we use can be found in[3]. While both the standard EM and the VBEM produce accurate abundance estimates, those produced by the VBEM seem, generally, to be a bit more accurate. Further, the VBEM tends to converge after fewer iterations, so it may result in a shorter runtime; especially if you are computing many bootstrap samples.

[1] Li, Bo, et al. "RNA-Seq gene expression estimation with read mapping uncertainty." Bioinformatics 26.4 (2010): 493-500.

[2] Nariai, Naoki, et al. "TIGAR: transcript isoform abundance estimation method with gapped alignment of RNA-Seq data by variational Bayesian inference." Bioinformatics (2013): btt381.

[3] Rob Patro, Geet Duggal & Carl Kingsford "Accurate, fast, and model-aware transcript expression quantification with Salmon" bioRxiv doi: http://dx.doi.org/10.1101/021592

--numBootstraps

Sailfish has the ability to optionally compute bootstrapped abundance estimates. This is done by resampling (with replacement) from the counts assigned to the fragment equivalence classes, and then re-running the optimization procedure, either the EM or VBEM, for each such sample. The values of these different bootstraps allows us to assess technical variance in the main abundance estimates we produce. Such estimates can be useful for downstream (e.g. differential expression) tools that can make use of such uncertainty estimates. This option takes a positive integer that dictates the number of bootstrap samples to compute. The more samples computed, the better the estimates of varaiance, but the more computation (and time) required.

--numGibbsSamples

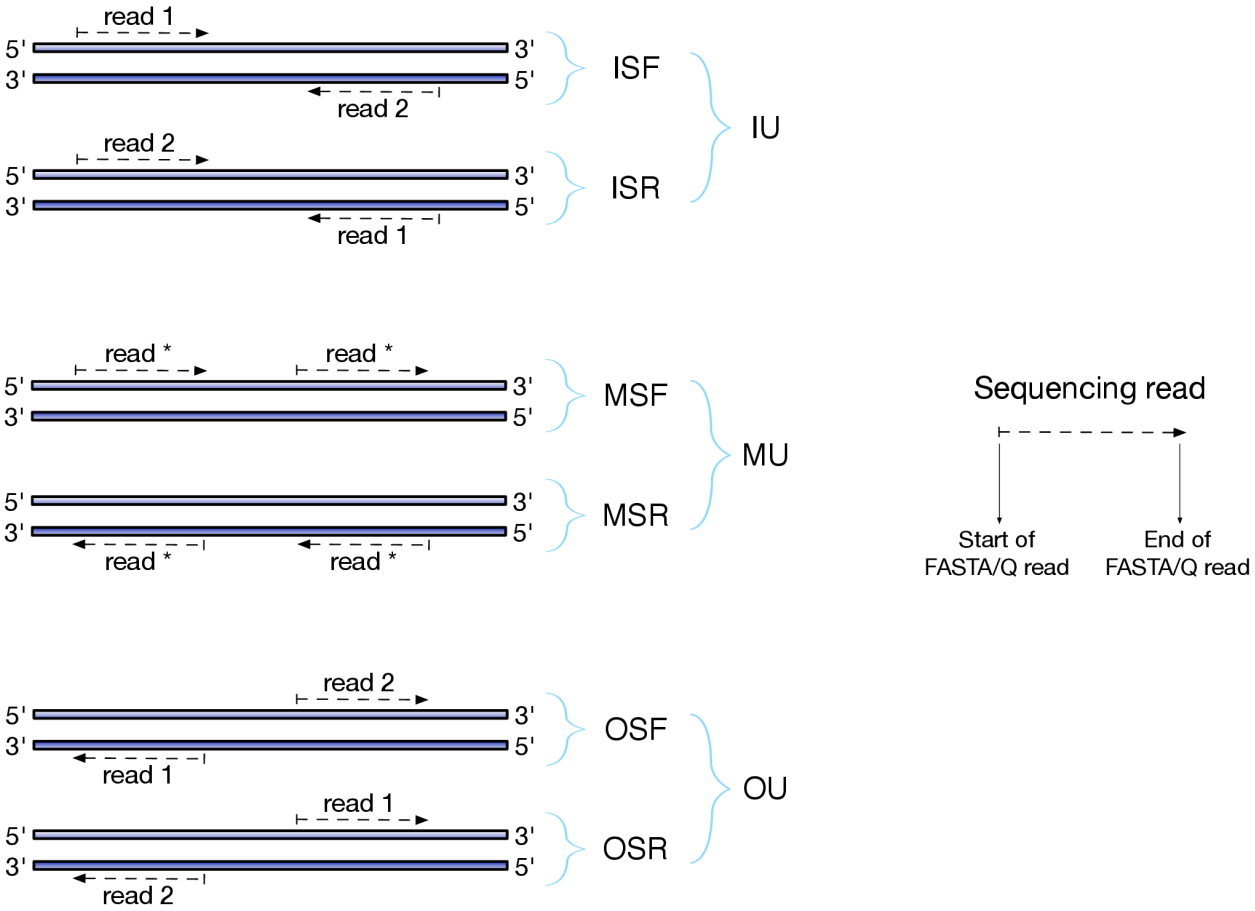
Just as with the bootstrap procedure above, this option produces samples that allow us to estimate the variance in abundance estimates. However, in this case the samples are generated using posterior Gibbs sampling over the fragment equivalence classes rather than bootstrapping. We are currently analyzing these different approaches to assess the potential trade-offs in time / accuracy. The `--numBootstraps` and `--numGibbsSamples` options are mutually exclusive (i.e. in a given run, you must set at most one of these options to a positive integer.)

References

CHAPTER 4

Fragment Library Types

There are numerous library preparation protocols for RNA-seq that result in sequencing reads with different characteristics. For example, reads can be single end (only one side of a fragment is recorded as a read) or paired-end (reads are generated from both ends of a fragment). Further, the sequencing reads themselves may be unstraned or strand-specific. Finally, paired-end protocols will have a specified relative orientation. To characterize the various different typs of sequencing libraries, we've created a miniature "language" that allows for the succinct description of the many different types of possible fragment libraries. For paired-end reads, the possible orientations, along with a graphical description of what they mean, are illustrated below:



The library type string consists of three parts: the relative orientation of the reads, the strandedness of the library, and the directionality of the reads.

The first part of the library string (relative orientation) is only provided if the library is paired-end. The possible options are:

```
I = inward
O = outward
M = matching
```

The second part of the read library string specifies whether the protocol is stranded or unstranded; the options are:

```
S = stranded
U = unstranded
```

If the protocol is unstranded, then we're done. The final part of the library string specifies the strand from which the read originates in a strand-specific protocol — it is only provided if the library is stranded (i.e. if the library format string is of the form S). The possible values are:

```
F = read 1 (or single-end read) comes from the forward strand
R = read 1 (or single-end read) comes from the reverse strand
```

So, for example, if you wanted to specify a fragment library of strand-specific paired-end reads, oriented toward each other, where read 1 comes from the forward strand and read 2 comes from the reverse strand, you would specify `-l ISF` on the command line. This designates that the library being processed has the type "ISF" meaning, **I**nward (the relative orientation), **S**tranded (the protocol is strand-specific), **F**orward (read 1 comes from the forward strand).

The single end library strings are a bit simpler than their pair-end counter parts, since there is no relative orientation of which to speak. Thus, the only possible library format types for single-end reads are `U` (for unstranded), `SF` (for strand-specific reads coming from the forward strand) and `SR` (for strand-specific reads coming from the reverse strand).

A few more examples of some library format strings and their interpretations are:

```
IU (an unstranded paired-end library where the reads face each other)
```

```
SF (a stranded single-end protocol where the reads come from the forward strand)
```

```
OSR (a stranded paired-end protocol where the reads face away from each other,
     read1 comes from reverse strand and read2 comes from the forward strand)
```

Note: Correspondence to TopHat library types

The popular TopHat RNA-seq read aligner has a different convention for specifying the format of the library. Below is a table that provides the corresponding sailfish/salmon library format string for each of the potential TopHat library types:

TopHat	Salmon (and Sailfish)	
	Paired-end	Single-end
`-fr-unstranded`	`-l IU`	`-l U`
`-fr-firststrand`	`-l ISR`	`-l SR`
`-fr-secondstrand`	`-l ISF`	`-l SF`

The remaining salmon library format strings are not directly expressible in terms of the TopHat library types, and so there is no direct mapping for them.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search