khmer Documentation

Release 1.0-44-g53a9785

2010-2014 Michael R. Crusoe, Greg Edvenson, Jordan Fish, Adina

June 10, 2014

Contents

Authors Michael R. Crusoe, Greg Edvenson, Jordan Fish, Adina Howe, Luiz Irber, Eric McDonald, Joshua Nahum, Kaben Nanlohy, Humberto Ortiz-Zuazaga, Jason Pell, Jared Simpson, Camille Scott, Ramakrishnan Rajaram Srinivasan, Qingpeng Zhang, and C. Titus Brown

Contact khmer-project@idyll.org

khmer is a library and suite of command line tools for working with DNA sequence. It is primarily aimed at short-read sequencing data such as that produced by the Illumina platform. khmer takes a k-mer-centric approach to sequence analysis, hence the name.

There are two mailing lists dedicated to khmer, an announcements-only list and a discussion list. To search their archives and sign-up for them, please visit the following URLs:

- Discussion: http://lists.idyll.org/listinfo/khmer
- Announcements: http://lists.idyll.org/listinfo/khmer-announce

The archives for the khmer list are available at: http://lists.idyll.org/pipermail/khmer/

khmer development has largely been supported by AFRI Competitive Grant no. 2010-65205-20361 from the USDA NIFA, and is now funded by the National Human Genome Research Institute of the National Institutes of Health under Award Number R01HG007513, both to C. Titus Brown.

Contents:

CHAPTER 1

Introduction to khmer

1.1 Introduction

khmer is a library and toolkit for doing k-mer-based dataset analysis and transformations. Our focus in developing it has been on scaling assembly of metagenomes and mRNA.

khmer can be used for a number of transformations, include inexact transformations (abundance filtering and error trimming) and exact transformations (graph-size filtering, to throw away disconnected reads; and partitioning, to split reads into disjoint sets). Of these, only partitioning is not constant memory. In all cases, the memory required for assembly with Velvet or another de Bruijn graph assembler will be more than the memory required to use our software. Our software will not increase the memory required for Velvet, either, although we may not be able to *decrease* the memory required for assembly for every data set.

Most of khmer relies on an underlying probabilistic data structure known as a Bloom filter (also see MinCount Sketch), which is essentially a set of hash tables, each of different size, with no collision detection. These hash tables are used to store the presence of specific k-mers and/or their count. The lack of collision detection means that the Bloom filter may report a k-mer as being "present" when it is not, in fact, in the data set; however, it will never incorrectly report a k-mer as being absent when it *is* present. This one-sided error makes the Bloom filter very useful for certain kinds of operations.

khmer is also independent of K, and currently works for K <= 32. We will be integrating code for up to K=64 soon.

khmer is implemented in C++ with a Python wrapper, which is what all of the scripts use.

1.2 Using khmer

khmer comes "out of the box" with a number of scripts that make it immediately useful for a few different operations, including:

- normalizing read coverage ("digital normalization")
- dividing reads into disjoint sets that do not connect ("partitioning")
- eliminating reads that will not be used by a de Bruijn graph assembler;
- removing reads with low- or high-abundance k-mers;
- trimming reads of certain kinds of sequencing errors;
- counting k-mers and estimating data set coverage based on k-mer counts;
- running Velvet and calculating assembly statistics;
- optimizing assemblies on various parameters;

- converting FASTA to FASTQ;

and a few other random functions.

1.3 Practical considerations

The most important thing to think about when using khmer is whether or not the transformation or filter you're applying is appropriate for the data you're trying to assemble. Two of the most powerful operations available in khmer, graph-size filtering and graph partitioning, only make sense for assembly datasets with many theoretically unconnected components. This is typical of metagenomic data sets.

The second most important consideration is memory usage. The effectiveness of all of the Bloom filter-based functions (which is everything interesting in khmer!) depends critically on having enough memory to do a good job. See *Choosing table sizes for khmer* for more information.

1.4 Copyright and license

CHAPTER 2

Installing and running khmer

You'll need Python 2.7+ and internet access.

The khmer project currently works with Python 2.6 but we target Python 2.7+.

2.1 Build requirements

2.1.1 OS X

If you just want to use the khmer project tools and not develop them then skip to step 4.

1. Install Xcode from the Mac App Store (requires root).
2. Register as an Apple Developer.
3. Install the Xcode command-line tools: Xcode -> preferences -> Downloads -> Command Line Tools (requires root).
4. From a terminal install virtualenv. You'll need the URL of the latest virtualenv release.

```
curl -O https://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.x.y.tar.gz
tar xzf virtualenv*
cd virtualenv-*; python virtualenv.py ../khmerEnv; cd ..
source khmerEnv/bin/activate
```

2.1.2 Linux

1. Install the python development environment, virtualenv, pip, and gcc.
   - On recent Debian and Ubuntu this can be done with:

     ```
     sudo apt-get install python2.7-dev python-virtualenv python-pip gcc
     ```

   - For RHEL6:

     ```
     sudo yum install -y python-devel python-pip git gcc gcc-c++ make
     sudo pip install virtualenv
     ```

2. Create a virtualenv and activate it:

```
cd a/writeable/directory/
virtualenv khmerEnv
source khmerEnv/bin/activate
```

Linux users without root access can try step 4 from the OS X instructions above.

2.2 Latest stable release

1. Use pip to download, build, and install khmer and its dependencies:

   ```
   pip install khmer
   ```

2. The scripts are now in the `env/bin` directory and ready for your use. You can directly use them by name, see *khmer's command-line interface*.

3. When returning to khmer after installing it you will need to reactivate the virtualenv first:

   ```
   source khmerEnv/bin/activate
   ```

2.3 Latest development branch

Repeat the above but modify the pip install line:

```
pip install git+https://github.com/ged-lab/khmer.git@master#egg=khmer
```

You can change `master` in the above command to the name of another branch.

2.3.1 Run the tests

If you're running a version of `pip` less than 1.4 and you want to run the tests then you should upgrade `pip`:

```
pip install --user --upgrade pip
```

Repeat the appropriate installation procedure from above but add "–no-clean" to the `pip` invocation.

The source will be in the `khmerEnv/build/khmer` directory. Run `make test` there.

CHAPTER 3

A few examples

See the 'examples' subdirectory for complete examples.

3.1 STAMPS data set

The 'stamps' data set is a fake metagenome-like data set containing two species, mixed at a 10:1 ratio. The source genomes are in 'data/stamps-genomes.fa'. The reads file is in 'data/stamps-reads.fa.gz', and consists of 100-base reads with a 1% error rate.

The example shows how to construct k-mer abundance histograms, as well as the effect of digital normalization and partitioning on the k-mer abundance distribution.

See the script for running everything and the IPython Notebook.

For an overall discussion and some slides to explain what's going on, visit the Web site for a 2013 HMP metagenome assembly webinar that Titus Brown gave.

CHAPTER 4

An assembly handbook for khmer - rough draft

date 2012-11-2

An increasing number of people are asking about using our assembly approaches for things that we haven't yet written (or posted) papers about. Moreover, our assembly strategies themselves are also under constant evolution as we do more research and find ever-wider applicability of our approaches.

Note, this is an exact copy of Titus' blog post, here – go check the bottom of that for comments.

4.1 Authors

This handbook distills the cumulative expertise of Adina Howe, Titus Brown, Erich Schwarz, Jason Pell, Camille Scott, Elijah Lowe, Kanchan Pavangadkar, Likit Preeyanon, and others.

4.2 Introduction

khmer is a general framework for low-memory k-mer counting, filtering, and advanced trickery.

The latest source is always available here.

khmer is really focused on short read data, and, more specifically, Illumina, because that's where we have a too-much-data problem. However, a lot of the prescriptions below can be adapted to longer read technologies such as 454 and Ion Torrent without much effort.

Don't try to use our k-mer approaches with PacBio – the error rate is too high.

There are currently two papers available on khmer: the partitioning paper and the digital normalization paper.

There are many blog posts about this stuff on Titus Brown's blog. We will try to link them in where appropriate.

4.3 Asking for help

There's some documentation here:

> https://khmer.readthedocs.org/en/latest/

There's also a khmer mailing list at lists.idyll.org that you can use to get help with khmer. To sign up, just go to the khmer lists page and subscribe.

4.4 Preparing your sequences

Do all the quality filtering, trimming, etc. that you think you should do.

Most of the khmer tools currently work "out of the box" on interleaved paired-end data. Ask on the list if you're not sure.

All of our scripts will take in .fq or .fastq files as FASTQ, and all other files as FASTA. gzip files are always accepted. Let us know if not; that's a bug!

Most scripts *output* FASTA, and some mangle headers. Sorry. We're working on outputting FASTQ for FASTQ input, and removing any header mangling.

4.5 Picking k-mer table sizes and k parameters

For k-mer table sizes, read *Choosing table sizes for khmer*

For k-mer sizes, we recommend k=20 for digital normalization and k=32 for partitioning; then assemble with a variety of k parameters.

4.6 Genome assembly, including MDA samples and highly polymorphic genomes

1. Apply digital normalization as follows.

Broadly, normalize each insert library separately, in the following way:

For high-coverage libraries (> ~50x), do three-pass digital normalization: run normalize-by-median to C=20 and then run filter-abund with C=1. Now split out the remaining paired-end/interleaved and single-end reads using strip-and-split-for-assembly, and normalize-by-median the paired-end and single-end files to C=5 (in that order).

For low-coverage libraries (< 50x) do single-pass digital normalization: run normalize-by-median to C=10.

2. Extract any remaining paired-end reads and lump remaining orphan reads into singletons using strip-and-split-for-assembly

3. Then assemble as normal, with appropriate insert size specs etc. for the paired end reads.

You can read about this process in the digital normalization paper.

4.7 mRNAseq assembly

1. Apply single-pass digital normalization.

Run normalize-by-median to C=20.

2. Extract any remaining paired-end reads and lump remaining orphan reads into singletons using strip-and-split-for-assembly

3. Then assemble as normal, with appropriate insert size specs etc. for the paired end reads.

You can read about this process in the digital normalization paper.

4.8 Metagenome assembly

1. Apply single-pass digital normalization.

Run normalize-by-median to C=20 (we've also found C=10 works fine).

2. Run filter-below-abund with C=50 (if you diginormed to C=10) or C=100 (if you diginormed to C=20);

3. Partition reads with load-graph, etc. etc.

4. Assemble groups as normal, extracting paired-end reads and lumping remaining orphan reads into singletons using strip-and-split-for-assembly.

(We actually use Velvet at this point, but there should be no harm in using a metagenome assembler such as MetaVelvet or MetaIDBA or SOAPdenovo.)

Read more about this in the partitioning paper. We have some upcoming papers on partitioning and metagenome assembly, too; we'll link those in when we can.

4.9 Metatranscriptome assembly

(Not tested by us!)

1. Apply single-pass digital normalization.

Run normalize-by-median to C=20.

2. Extract any remaining paired-end reads and lump remaining orphan reads into singletons using strip-and-split-for-assembly

3. Then assemble with a genome or metagenome assembler, *not* an mRNAseq assembler. Use appropriate insert size specs etc. for the paired end reads.

4.10 Preprocessing Illumina for other applications

(Not tested by us!)

Others have told us that you can apply digital normalization to Illumina data prior to using Illumina for RNA scaffolding or error correcting PacBio reads.

Our suggestion for this, based on no evidence whatsoever, is to diginorm the Illumina data to C=20.

4.11 Quantifying mRNAseq or metagenomes assembled with digital normalization

For now, khmer only deals with assembly! So: assemble. Then, go back to your original, unnormalized reads, and map those to your assembly with e.g. bowtie. Then count as you normally would :).

4.12 Philosophy of digital normalization

The basic philosophy of digital normalization is "load your most valuable reads first." Diginorm gets rid of redundancy iteratively, so you are more likely to retain the first reads fed in; this means you should load in paired end reads, or longer reads, first.

4.13 Iterative and independent normalization

You can use *--loadtable* and *--savetable* to do iterative normalizations on multiple files in multiple steps. For example, break

```
normalize-by-median.py [ ... ] file1.fa file2.fa file3.fa
```

into multiple steps like so:

```
normalize-by-median.py [ ... ] --savetable file1.kh file1.fa
normalize-by-median.py [ ... ] --loadtable file1.kh --savetable file2.kh file2.fa
normalize-by-median.py [ ... ] --loadtable file2.kh --savetable file3.kh file3.fa
```

The results should be identical!

If you want to independently normalize multiple files for speed reasons, go ahead. Just remember to do a combined normalization at the end. For example, instead of

```
normalize-by-median.py [ ... ] file1.fa file2.fa file3.fa
```

you could do

```
normalize-by-median.py [ ... ] file1.fa
normalize-by-median.py [ ... ] file2.fa
normalize-by-median.py [ ... ] file3.fa
```

and then do a final

```
normalize-by-median.py [ ... ] file1.fa.keep file2.fa.keep file3.fa.keep
```

The results will not be identical, but should not differ significantly. The multipass approach will take more total time but may end up being faster walltime because you can execute the independent normalizations on multiple computers.

For a cleverer approach that we will someday implement, read the Beachcomber's Dilemma.

4.14 Validating and comparing assemblies

More here soon :).

CHAPTER 5

khmer's command-line interface

The simplest way to use khmer's functionality is through the command line scripts, located in the scripts/ directory of the khmer distribution. Below is our documentation for these scripts. Note that all scripts can be given `-h` which will print out a list of arguments taken by that script.

Many scripts take `-x` and `-N` parameters, which drive khmer's memory usage. These parameters depend on details of your data set; for more information on how to choose them, see *Choosing table sizes for khmer*.

You can also override the default values of `--ksize`/`-k`, `--n_tables`/`-N`, and `--min-tablesize`/`-x` with the environment variables *KHMER_KSIZE*, *KHMER_N_TABLES*, and *KHMER_MIN_TABLESIZE* respectively.

1. *k-mer counting and abundance filtering*
2. *Partitioning*
3. *Digital normalization*
4. *Read handling: interleaving, splitting, etc.*

Note: Almost all scripts take in either FASTA and FASTQ format, and output the same. Some scripts may only recognize FASTQ if the file ending is '.fq' or '.fastq', at least for now.

Files ending with '.gz' will be treated as gzipped files, and files ending with '.bz2' will be treated as bzip2'd files.

5.1 k-mer counting and abundance filtering

5.1.1 load-into-counting.py

Build a k-mer counting table from the given sequences.

usage: load-into-counting.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads N_THREADS] [-b] output_countingtable_filename input_sequence_filename [input_sequence_filename ...]

output_countingtable_filename
: The name of the file to write the k-mer counting table to.

input_sequence_filename
: The names of one or more FAST[AQ] input sequence files.

-h, --help
: show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

-b, --no-bigcount
Do not count k-mers past 255

Note: with *-b* the output will be the exact size of the k-mer counting table and this script will use a constant amount of memory. In exchange k-mer counts will stop at 255. The memory usage of this script with *-b* will be about 1.15x the product of the *-x* and *-N* numbers.

Example:

```
load_into_counting.py -k 20 -x 5e7 out.kh data/100k-filtered.fa
```

Multiple threads can be used to accelerate the process, if you have extra cores to spare.

Example:

```
load_into_counting.py -k 20 -x 5e7 -T 4 out.kh data/100k-filtered.fa
```

5.1.2 abundance-dist.py

Calculate abundance distribution of the k-mers in the sequence file using a pre-made k-mer counting table.

usage: abundance-dist.py [-h] [-z] [-s] [–version] input_counting_table_filename input_sequence_filename output_histogram_filename

input_counting_table_filename
The name of the input k-mer counting table file.

input_sequence_filename
The name of the input FAST[AQ] sequence file.

output_histogram_filename
The columns are: (1) k-mer abundance, (2) k-mer count, (3) cumulative count, (4) fraction of total distinct k-mers.

-h, --help
show this help message and exit

-z, --no-zero
Do not output 0-count bins

-s, --squash
Overwrite output file if it exists

--version
show program's version number and exit

5.1.3 abundance-dist-single.py

Calculate the abundance distribution of k-mers from a single sequence file.

usage: abundance-dist-single.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads THREADS] [-z] [-b] [-s] [–savetable filename] input_sequence_filename output_histogram_filename

input_sequence_filename
The name of the input FAST[AQ] sequence file.

output_histogram_filename
The name of the output histogram file. The columns are: (1) k-mer abundance, (2) k-mer count, (3) cumulative count, (4) fraction of total distinct k-mers.

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

-z, --no-zero
Do not output 0-count bins

-b, --no-bigcount
Do not count k-mers past 255

-s, --squash
Overwrite output file if it exists

--savetable <filename>
Save the k-mer counting table to the specified filename.

Note that with *-b* this script is constant memory; in exchange, k-mer counts will stop at 255. The memory usage of this script with *-b* will be about 1.15x the product of the *-x* and *-N* numbers.

To count k-mers in multiple files use **load_into_counting.py** and **abundance_dist.py**.

5.1.4 filter-abund.py

Trim sequences at a minimum k-mer abundance.

usage: filter-abund.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads THREADS] [–cutoff CUTOFF] [–variable-coverage] [–normalize-to NORMALIZE_TO] [-o optional_output_filename] input_presence_table_filename input_sequence_filename [input_sequence_filename ...]

input_presence_table_filename
The input k-mer presence table filename

input_sequence_filename
Input FAST[AQ] sequence filename

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

--cutoff <int>, **-C** <int>
Trim at k-mers below this abundance.

--variable-coverage, -V
Only trim low-abundance k-mers from sequences that have high coverage.

--normalize-to <int>, **-Z** <int>
Base the variable-coverage cutoff on this median k-mer abundance.

-o <optional_output_filename>, **--out** <optional_output_filename>
Output the trimmed sequences into a single file with the given filename instead of creating a new file for each input file.

Trimmed sequences will be placed in ${input_sequence_filename}.abundfilt for each input sequence file. If the input sequences are from RNAseq or metagenome sequencing then *--variable-coverage* should be used.

Example:

```
load-into-counting.py -k 20 -x 5e7 table.kh data/100k-filtered.fa
filter-abund.py -C 2 table.kh data/100k-filtered.fa
```

5.1.5 filter-abund-single.py

Trims sequences at a minimum k-mer abundance (in memory version).

usage: filter-abund-single.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads THREADS] [–cutoff CUTOFF] [–savetable filename] input_sequence_filename

input_sequence_filename
FAST[AQ] sequence file to trim

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

--cutoff <int>, **-C** <int>
Trim at k-mers below this abundance.

--savetable <filename>
If present, the name of the file to save the k-mer counting table to

Trimmed sequences will be placed in ${input_sequence_filename}.abundfilt.

This script is constant memory.

To trim reads based on k-mer abundance across multiple files, use **load-into-counting.py** and **filter-abund.py**.

Example:

```
filter-abund-single.py -k 20 -x 5e7 -C 2 data/100k-filtered.fa
```

5.1.6 count-median.py

Count k-mers summary stats for sequences

usage: count-median.py [-h] [–version] input_counting_table_filename input_sequence_filename output_summary_filename

input_counting_table_filename
input k-mer count table filename

input_sequence_filename
input FAST[AQ] sequence filename

output_summary_filename
output summary filename

-h, --help
show this help message and exit

--version
show program's version number and exit

Count the median/avg k-mer abundance for each sequence in the input file, based on the k-mer counts in the given k-mer counting table. Can be used to estimate expression levels (mRNAseq) or coverage (genomic/metagenomic).

The output file contains sequence id, median, average, stddev, and seq length.

NOTE: All 'N's in the input sequences are converted to 'G's.

5.1.7 count-overlap.py

Count the overlap k-mers which are the k-mers appearing in two sequence datasets.

usage: count-overlap.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] input_presence_table_filename input_sequence_filename output_report_filename

input_presence_table_filename
input k-mer presence table filename

input_sequence_filename
input sequence filename

output_report_filename
output report filename

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

An additional report will be written to ${output_report_filename}.curve containing the increase of overlap k-mers as the number of sequences in the second database increases.

5.2 Partitioning

5.2.1 do-partition.py

Load, partition, and annotate FAST[AQ] sequences

usage: do-partition.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–subset-size SUBSET_SIZE] [–no-big-traverse] [–threads N_THREADS] [–keep-subsets] graphbase input_sequence_filename [input_sequence_filename ...]

graphbase
base name for output files

input_sequence_filename
input FAST[AQ] sequence filenames

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--subset-size <float>, **-s** <float>
Set subset size (usually 1e5-1e6 is good)

--no-big-traverse
Truncate graph joins at big traversals

--threads, -T
Number of simultaneous threads to execute

--keep-subsets
Keep individual subsets (default: False)

Load in a set of sequences, partition them, merge the partitions, and annotate the original sequences files with the partition information.

This script combines the functionality of **load-graph.py**, **partition-graph.py**, **merge-partitions.py**, and **annotate-partitions.py** into one script. This is convenient but should probably not be used for large data sets, because **do-partition.py** doesn't provide save/resume functionality.

5.2.2 load-graph.py

Load sequences into the compressible graph format plus optional tagset.

usage: load-graph.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–threads N_THREADS] [–no-build-tagset] output_presence_table_filename input_sequence_filename [input_sequence_filename ...]

output_presence_table_filename
output k-mer presence table filename.

input_sequence_filename
input FAST[AQ] sequence filename

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

--no-build-tagset, -n
Do NOT construct tagset while loading sequences

See **extract-partitions.py** for a complete workflow.

5.2.3 partition-graph.py

Partition a sequence graph based upon waypoint connectivity

usage: partition-graph.py [-h] [–stoptags filename] [–subset-size SUBSET_SIZE] [–no-big-traverse] [–version] [–threads THREADS] basename

basename
basename of the input k-mer presence table + tagset files

-h, --help
show this help message and exit

--stoptags <filename>, **-S** <filename>
Use stoptags in this file during partitioning

--subset-size <float>, **-s** <float>
Set subset size (usually 1e5-1e6 is good)

--no-big-traverse
Truncate graph joins at big traversals

--version
show program's version number and exit

--threads <int>, **-T** <int>
Number of simultaneous threads to execute

The resulting partition maps are saved as '${basename}.subset.#.pmap' files.

See 'Artifact removal' to understand the stoptags argument.

5.2.4 merge-partition.py

Merge partition map '.pmap' files.

usage: merge-partition.py [-h] [–ksize KSIZE] [–keep-subsets] [–version] graphbase

graphbase
basename for input and output files

-h, --help
show this help message and exit

--ksize <int>, **-k** <int>
k-mer size (default: 32)

--keep-subsets
Keep individual subsets (default: False)

--version
show program's version number and exit

Take the ${graphbase}.subset.#.pmap files and merge them all into a single ${graphbase}.pmap.merged file for **annotate-partitions.py** to use.

5.2.5 annotate-partitions.py

Annotate sequences with partition IDs.

usage: annotate-partitions.py [-h] [–ksize KSIZE] [–version] graphbase input_sequence_filename [input_sequence_filename ...]

graphbase
basename for input and output files

input_sequence_filename
input FAST[AQ] sequences to annotate.

-h, --help
show this help message and exit

--ksize <int>, **-k** <int>
k-mer size (default: 32)

--version
show program's version number and exit

Load in a partitionmap (generally produced by partition-graph.py or merge-partitions.py) and annotate the sequences in the given files with their partition IDs. Use **extract-partitions.py** to extract sequences into separate group files.

Example (results will be in `random-20-a.fa.part`):

```
load-graph.py -k 20 example tests/test-data/random-20-a.fa
partition-graph.py example
merge-partitions.py -k 20 example
annotate-partitions.py -k 20 example tests/test-data/random-20-a.fa
```

5.2.6 extract-partitions.py

Separate sequences that are annotated with partitions into grouped files.

usage: extract-partitions.py [-h] [–max-size MAX_SIZE] [–min-partition-size MIN_PART_SIZE] [–no-output-groups] [–output-unassigned] [–version] output_filename_prefix input_partition_filename [input_partition_filename ...]

output_filename_prefix

input_partition_filename

-h, --help
show this help message and exit

--max-size <int>, **-X** <int>
Max group size (n sequences)

--min-partition-size <int>, **-m** <int>
Minimum partition size worth keeping

--no-output-groups, -n
Do not actually output groups files.

--output-unassigned, -U
Output unassigned sequences, too

--version
show program's version number and exit

Example (results will be in `example.group0000.fa`):

```
load-graph.py -k 20 example tests/test-data/random-20-a.fa
partition-graph.py example
merge-partitions.py -k 20 example
annotate-partitions.py -k 20 example tests/test-data/random-20-a.fa
extract-partitions.py example random-20-a.fa.part
```

5.2.7 Artifact removal

The following scripts are specialized scripts for finding and removing highly-connected k-mers (HCKs). See *Partitioning large data sets (50m+ reads)*.

make-initial-stoptags.py

Find an initial set of highly connected k-mers.

usage: make-initial-stoptags.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–subset-size SUBSET_SIZE] [–stoptags filename] graphbase

graphbase
basename for input and output filenames

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

--subset-size <float>, **-s** <float>
Set subset size (default 1e4 is prob ok)

--stoptags <filename>, **-S** <filename>
Use stoptags in this file during partitioning

Loads a k-mer presence table/tagset pair created by load-graph.py, and does a small set of traversals from graph waypoints; on these traversals, looks for k-mers that are repeatedly traversed in high-density regions of the graph, i.e. are highly connected. Outputs those k-mers as an initial set of stoptags, which can be fed into partition-graph.py, find-knots.py, and filter-stoptags.py.

The k-mer counting table size options parameters are for a k-mer counting table to keep track of repeatedly-traversed k-mers. The subset size option specifies the number of waypoints from which to traverse; for highly connected data sets, the default (1000) is probably ok.

find-knots.py

Find all highly connected k-mers.

usage: find-knots.py [-h] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [–version] graphbase

graphbase
Basename for the input and output files.

-h, --help
show this help message and exit

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on the size of the k-mer counting table(s)

--version
show program's version number and exit

Load an k-mer presence table/tagset pair created by load-graph, and a set of pmap files created by partition-graph. Go through each pmap file, select the largest partition in each, and do the same kind of traversal as in **make-initial-stoptags.py** from each of the waypoints in that partition; this should identify all of the HCKs in that partition. These HCKs are output to <graphbase>.stoptags after each pmap file.

Parameter choice is reasonably important. See the pipeline in *Partitioning large data sets (50m+ reads)* for an example run.

This script is not very scalable and may blow up memory and die horribly. You should be able to use the intermediate stoptags to restart the process, and if you eliminate the already-processed pmap files, you can continue where you left off.

filter-stoptags.py

Trim sequences at stoptags.

usage: filter-stoptags.py [-h] [–ksize KSIZE] [–version] input_stoptags_filename input_sequence_filename [input_sequence_filename ...]

input_stoptags_filename

input_sequence_filename

-h, --help
show this help message and exit

--ksize <int>, **-k** <int>
k-mer size

--version
show program's version number and exit

Load stoptags in from the given .stoptags file and use them to trim or remove the sequences in <file1-N>. Trimmed sequences will be placed in <fileN>.stopfilt.

5.3 Digital normalization

5.3.1 normalize-by-median.py

Do digital normalization (remove mostly redundant sequences)

usage: normalize-by-median.py [-h] [–version] [-q] [–ksize KSIZE] [–n_tables N_TABLES] [–min-tablesize MIN_TABLESIZE] [-C CUTOFF] [-p] [-s filename] [-R filename] [-f] [–save-on-failure] [-d DUMP_FREQUENCY] [-o filename] [-l filename] input_sequence_filename [input_sequence_filename ...]

input_sequence_filename
Input FAST[AQ] sequence filename.

-h, --help
show this help message and exit

--version
show program's version number and exit

-q, --quiet

--ksize <int>, **-k** <int>
k-mer size to use

--n_tables <int>, **-N** <int>
number of k-mer counting tables to use

--min-tablesize <float>, **-x** <float>
lower bound on tablesize to use

-C <int>, **--cutoff** <int>

-p, --paired

-s <filename>, **--savetable** <filename>

-R <filename>, **--report** <filename>

-f, --fault-tolerant
continue on next file if read errors are encountered

--save-on-failure
Save k-mer counting table when an error occurs

-d <int>, **--dump-frequency** <int>
dump k-mer counting table every d files

-o <filename>, **--out** <filename>
only output a single file with the specified filename

-l <filename>, **--loadtable** <filename>
load a precomputed k-mer table from disk

Discard sequences based on whether or not their median k-mer abundance lies above a specified cutoff. Kept sequences will be placed in <fileN>.keep.

Paired end reads will be considered together if *-p* is set. If either read will be kept, then both will be kept. This should result in keeping (or discarding) each sequencing fragment. This helps with retention of repeats, especially.

With *-s*/*--savetable*, the k-mer counting table will be saved to the specified file after all sequences have been processed. With *-d*, the k-mer counting table will be saved every d files for multifile runs; if *-s* is set, the specified name will be used, and if not, the name *backup.ct* will be used. *-l*/*--loadtable* will load the specified k-mer counting table before processing the specified files. Note that these tables are are in the same format as those produced by **load-into-counting.py** and consumed by **abundance-dist.py**.

-f/*--fault-tolerant* will force the program to continue upon encountering a formatting error in a sequence file; the k-mer counting table up to that point will be dumped, and processing will continue on the next file.

Example:

```
normalize-by-median.py -k 17 tests/test-data/test-abund-read-2.fa
```

Example:

```
normalize-by-median.py -p -k 17 tests/test-data/test-abund-read-paired.fa
```

Example:

```
normalize-by-median.py -k 17 -f tests/test-data/test-error-reads.fq tests/test-data/test-fastq-reads
```

Example:

```
normalize-by-median.py -k 17 -d 2 -s test.ct tests/test-data/test-abund-read-2.fa tests/test-data/tes
```

5.4 Read handling: interleaving, splitting, etc.

5.4.1 extract-paired-reads.py

Take a mixture of reads and split into pairs and orphans.

usage: extract-paired-reads.py [-h] [–version] infile

infile

-h, --help
show this help message and exit

--version
show program's version number and exit

The output is two files, <input file>.pe and <input file>.se, placed in the current directory. The .pe file contains interleaved and properly paired sequences, while the .se file contains orphan sequences.

Many assemblers (e.g. Velvet) require that you give them either perfectly interleaved files, or files containing only single reads. This script takes files that were originally interleaved but where reads may have been orphaned via error filtering, application of abundance filtering, digital normalization in non-paired mode, or partitioning.

Example:

```
extract-paired-reads.py tests/test-data/paired.fq
```

5.4.2 interleave-reads.py

Produce interleaved files from R1/R2 paired files

usage: interleave-reads.py [-h] [-o filename] [–version] infiles [infiles ...]

infiles

-h, --help
show this help message and exit

-o <filename>**, --output** <filename>

--version
show program's version number and exit

The output is an interleaved set of reads, with each read in <R1> paired with a read in <R2>. By default, the output goes to stdout unless *-o*/*--output* is specified.

As a "bonus", this file ensures that read names are formatted in a consistent way, such that they look like the pre-1.8 Casava format (@name/1, @name/2).

Example:

```
interleave-reads.py tests/test-data/paired.fq.1 tests/test-data/paired.fq.2 -o paired.fq
```

5.4.3 split-paired-reads.py

Split interleaved reads into two files, left and right.

usage: split-paired-reads.py [-h] [–version] infile

infile

-h, --help
show this help message and exit

--version
show program's version number and exit

Some programs want paired-end read input in the One True Format, which is interleaved; other programs want input in the Insanely Bad Format, with left- and right- reads separated. This reformats the former to the latter.

Example:

```
split-paired-reads.py tests/test-data/paired.fq
```

CHAPTER 6

Blog posts and additional documentation

6.1 Hashtable and filtering

The basic inexact-matching approach used by the hashtable code is described in this blog post:

http://ivory.idyll.org/blog/jul-10/kmer-filtering

A test data set (soil metagenomics, 88m reads, 10gb) is here:

http://angus.ged.msu.edu.s3.amazonaws.com/88m-reads.fa.gz

6.2 Illumina read abundance profiles

khmer can be used to look at systematic variations in k-mer statistics across Illumina reads; see, for example, this blog post:

http://ivory.idyll.org/blog/jul-10/illumina-read-phenomenology

The fasta-to-abundance-hist and abundance-hist-by-position scripts can be used to generate the k-mer abundance profile data, after loading all the k-mer counts into a .kh file:

```
# first, load all the k-mer counts:
load-into-counting.py -k 20 -x 1e7 25k.kh data/25k.fq.gz

# then, build the '.freq' file that contains all of the counts by position
python sandbox/fasta-to-abundance-hist.py 25k.kh data/25k.fq.gz

# sum across positions.
python sandbox/abundance-hist-by-position.py data/25k.fq.gz.freq > out.dist
```

The hashtable method 'dump_kmers_by_abundance' can be used to dump high abundance k-mers, but we don't have a script handy to do that yet.

You can assess high/low abundance k-mer distributions with the hi-lo-abundance-by-position script:

```
load-into-counting.py -k 20 25k.kh data/25k.fq.gz
python sandbox/hi-lo-abundance-by-position.py 25k.kh data/25k.fq.gz
```

This will produce two output files, <filename>.pos.abund=1 and <filename>.pos.abund=255.

CHAPTER 7

Choosing table sizes for khmer

If you look at the documentation for the scripts (*khmer's command-line interface*) you'll see two mysterious parameters – `-N` and `-x`, or, more verbosely, `-n_tables` and `--tablesize`. What are these, and how do you specify them?

7.1 The really short version

There is no way (except for experience, rules of thumb, and intuition) to know what these parameters should be up front. So, make the product of these two parameters be the size of your available memory:

```
-N 4 -x 4e9
```

for a machine with 16 GB of free memory, for example. Also see the rules of thumb, below.

7.2 The short version

These parameters specify the maximum memory usage of the primary data structure in khmer, which is basically N big hash tables of size x. The **product** of the number of hash tables and the size of the hash tables specifies the total amount of memory used.

This table is used to track k-mers. If it is too small, khmer will fail in various ways (and should complain), but there is no harm in making it too large. So, **the absolute safest thing to do is to specify as much memory as is available**. Most scripts will inform you of the total memory usage, and (at the end) will complain if it's too small.

For normalize-by-median, khmer uses one byte per hash entry, so: if you had 16 GB of available RAM, you should specify something like `-N 4 -x 4e9`, which multiplies out to about 16 GB.

For the graph partitioning stuff, khmer uses only 1 bit per k-mer, so you can multiple your available memory by 8: for 16 GB of RAM, you could use

```
-N 4 -x 32e9
```

which multiplies out to 128 Gbits of RAM, or 16 Gbytes.

Life is a bit more complicated than this, however, because some scripts – load-into-counting and load-graph – keep ancillary information that will consume memory beyond this table data structure. So if you run out of memory, decrease the table size.

Also see the rules of thumb, below.

7.3 The real full version

khmer's scripts, at their heart, represents k-mers in a very memory efficient way by taking advantage of two data structures, Bloom filters and CountMin Sketches, that are both *probabilistic* and *constant memory*. The "probabilistic" part means that there are false positives: the less memory you use, the more likely it is that khmer will think that k-mers are present when they are not, in fact, present.

Digital normalization (normalize-by-median and filter-abund) uses the CountMin Sketch data structure.

Graph partitioning (load-graph etc.) uses the Bloom filter data structure.

The practical ramifications of this are pretty cool. For example, your digital normalization is guaranteed not to increase in memory utilization, and graph partitioning is estimated to be 10-20x more memory efficient than any other de Bruijn graph representation. And hash tables (which is what Bloom filters and CountMin Sketches use) are really fast and efficient. Moreover, the optimal memory size for these primary data structures is dependent on the number of k-mers, but not explicitly on the size of k itself, which is very unusual.

In exchange for this memory efficiency, however, you gain a certain type of parameter complexity. Unlike your more typical k-mer package (like the Velvet assembler, or Jellyfish or Meryl or Tallymer), you are either guaranteed not to run out of memory (for digital normalization) or much less likely to do so (for partitioning).

The biggest problem with khmer is that there is a minimum hash number and size that you need to specify for a given number of k-mers, and you cannot confidently predict what it is before actually loading in the data. This, by the way, is also true for de Bruijn graph assemblers and all the other k-mer-based software – the final memory usage depends on the total number of k-mers, which in turn depends on the true size of your underlying genomic variation (e.g. genome or transcriptome size), the number of errors, and the k-mer size you choose (the k parameter) [see Conway & Bromage, 2011]. **The number of reads or the size of your data set is only somewhat correlated with the total number of k-mers.** Trimming protocols, sequencing depth, and polymorphism rates are all important factors that affect k-mer count.

The bad news is that we don't have good ways to estimate total k-mer count a priori, although we can give you some rules of thumb, below. In fact, counting the total number of distinct k-mers is a somewhat annoying challenge. Frankly, we recommend *just guessing* instead of trying to be all scientific about it.

The good news is that you can never give khmer too much memory! k-mer counting and set membership simply gets more and more accurate as you feed it more memory. (Although there may be performance hits from memory I/O, e.g. see the NUMA architecture.) The other good news is that khmer can measure the false positive rate and detect dangerously low memory conditions. For partitioning, we actually *know* what a too-high false positive rate is – our k-mer percolation paper lays out the math. For digital normalization, we assume that a false positive rate of 10% is bad. In both cases the data-loading scripts will exit with an error-code.

7.3.1 Rules of thumb

Just use -N 4, always, and vary the -x parameter.

For digital normalization, we recommend:

- `-x 2e9` for any amount of sequencing for a single microbial genome, MDA-amplified or single colony.
- `-x 4e9` for up to a billion mRNAseq reads from any organism. Past that, increase it.
- `-x 8e9` for most eukaryotic genome samples.
- `-x 8e9` will also handle most "simple" metagenomic samples (HMP on down)
- For metagenomic samples that are more complex, such as soil or marine, start as high as possible. For example, we are using `-x 64e9` for ~300 Gbp of soil reads.

For partitioning of complex metagenome samples, we recommend starting as high as you can – something like half your system memory. So if you have 256 GB of RAM, use `-N 4 -x 256e9` which will use 4 x 256 / 8 = 128 GB of RAM for the basic graph storage, leaving other memory for the ancillary data structures.

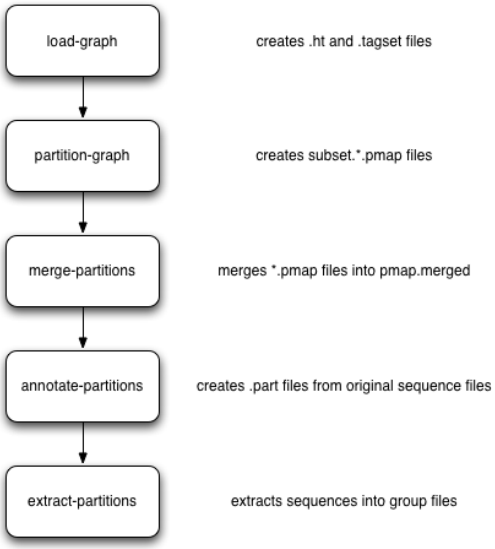
CHAPTER 8

Partitioning large data sets (50m+ reads)

"Partitioning" is what khmer calls the process of separating reads that do not connect to each other into different logical bins. The goal of partitioning is to apply divide & conquer to the process of metagenomic assembly.

8.1 Basic partitioning

The basic workflow for partitioning is in the figure below:

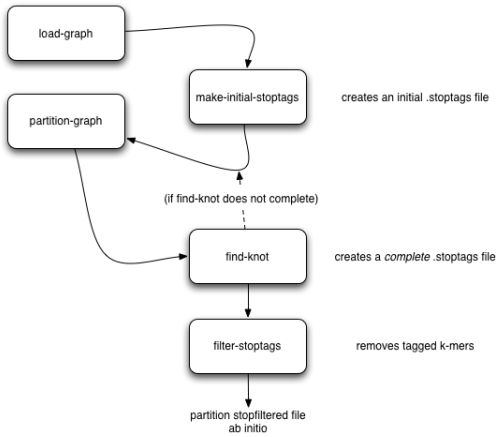


Briefly, you load everything into khmer's probabilistic graph representation; exhaustively explore the graph to find all disconnected sequences; merge the results of the (parallelized) graph exploration; annotate sequences with their partition; and then extract the different partitions into files grouped by partition size. These groups can then be assembled individually.

8.2 Artifact removal

As part of our partitioning research, we discovered that large Illumina data sets tend to contain a single large, connected component. This connected component seems to stem from sequencing artifacts that causes knots in the assembly graph. We have developed tools to forcibly remove the knot at the heart of the graph.

Here's the workflow:



8.3 Running on an example data set

Here is a set of commands for running both basic partitioning and artifact removal on a small soil metagenomics data set that we've made available for this purpose.

The data set is about 1.1G and you can download it from here:

https://s3.amazonaws.com/public.ged.msu.edu/khmer/iowa-corn-50m.fa.gz

```
cd /path/to/data

# the next command will create a '50m.ct' and a '50m.tagset',
# representing the de Bruijn graph
load-graph.py -k 32 -N 4 -x 16e9 50m iowa-corn-50m.fa.gz

# this will then partition that graph. should take a while.
# update threads to something higher if you have more cores.
# this creates a bunch of files, 50m.subset.*.pmap
partition-graph.py --threads 4 -s 1e5 50m

# now, merge the pmap files into one big pmap file, 50m.pmap.merged
merge-partitions.py 50m

# next, annotate the original sequences with their partition numbers.
# this will create iowa-corn-50m.fa.gz.part
annotate-partitions.py 50m iowa-corn-50m.fa.gz

# now, extract the partitions in groups into 'iowa-corn-50m.groupNNNN.fa'
extract-partitions.py iowa-corn-50m iowa-corn-50m.fa.gz.part

# at this point, you can assemble the group files individually.  Note,
# however, that the last one them is quite big?  this is because it's
# the lump! yay!

# if you want to break up the lump, go through the partitioning bit
# on the group file, but this time with a twist:
mv iowa-corn-50m.group0005.fa corn-50m.lump.fa

# create graph,
load-graph.py -x 8e9 lump corn-50m.lump.fa
```

```
# create an initial set of stoptags to help in knot-traversal; otherwise,
# partitioning and knot-traversal (which is systematic) is really expensive.
make-initial-stoptags.py lump

# now partition the graph, using the stoptags file
partition-graph.py --stoptags lump.stoptags lump

# use the partitioned subsets to find the k-mers that nucleate the lump
find-knots.py -x 2e8 -N 4 lump

# remove those k-mers from the fasta files
filter-stoptags.py *.stoptags corn-50m.lump.fa

# now, reload the filtered data set in and partition again.
# NOTE: 'load-graph.py' uses the file extension to determine
# if the file is formatted as FASTA or FASTQ. The default is
# fasta, therefore if your files are fastq formatted you need
# to append 'fastq' to the name so that 'load-graph.py'
# will parse the file correctly
load-graph.py -x 8e9 lumpfilt corn-50m.lump.fa.stopfilt
partition-graph.py -T 4 lumpfilt
merge-partitions.py lumpfilt
annotate-partitions.py lumpfilt corn-50m.lump.fa.stopfilt
extract-partitions.py corn-50m-lump corn-50m.lump.fa.stopfilt.part

# and voila, after all that, you should now have your de-knotted lump in
# corn-50m-lump.group*.fa.  The *.group????.fa files can now be
# assembled individually by your favorite assembler.
```

8.4 Post-partitioning assembly

The 'extract-partitions' script takes reads belonging to each partition and aggregates them into 'group' files; each group file contains at least one entire partition (and generally a lot more). Note, you can control the number of reads in each file (equiv, the size of these files) with some of the arguments that 'extract-partitions' takes.

Now that you have these files... what do you do with them? The short answer is: assemble them! Each of these group files contains reads that do not connect to reads in other files, so the files can be assembled individually (which is the whole point of partitioning).

If you're using Velvet, checkout the `sandbox/velvet-assemble.sh` script, which you can run like this:

```
bash /path/to/khmer/sandbox/velvet-assemble.sh <groupfile> <k>
```

This script does three things:

- first, it breaks the reads up into paired reads and single reads, and puts them in separate files (.pe and .se);
- second, it strips off the partition information from the reads, which confuses Velvet;
- and third, it runs velveth and velvetg to actually assemble.

You can implement your own approach, of course, but this is an example of what we do ourselves.

CHAPTER 9

Architecture and Design

What follows is an attempt to describe the overall architecture and design of the *khmer* software under the hood. Where appropriate, implementation details will be mentioned. Also, possible future directions and design considerations will be mentioned as appropriate.

9.1 Overview

Data pumps stage data from disk storage into an in-memory cache. The in-memory cache is divided into segments, one segment per thread. A cache manager exposes an interface for staging the data via the data pumps and for accessing the data in the cache segments. Read parsers convert the staged data into read objects. A separate state object is maintained for each thread using a parser. Existing-tracking or counting Bloom filters can use the read parsers as a source of reads from which to extract k-mers.

The read parsers and the layers under them can be controlled via global configuration objects, which provide default values during their instantiation. In many cases, these default values can also be overridden by supplying pertinent arguments to the constructors. Only one global configuration object is considered active at a given time; but, a singleton pattern is not enforced and more than one may be available to supply alternative configurations.

The top-level makefile for the project contains a user-configurable section, wherein preprocessor, compiler, and linker options may be selected via convenient, prefabricated bundles. The ability to generate profiling instrumentation, compile with debugging symbols, and to generate tracing instrumentation are all controlled via these option bundles. The lower levels of the code, such as the data pumps, cache manager, and read parsers all have significant built-in profiling and tracing instrumentation. This instrumentation is conditionally-compiled according to the option bundles selected in the top-level makefile.

9.2 Namespace

Unless otherwise noted, all C++ classes, functions, and static variables noted in this document are members of the `khmer` namespace. Likewise, unless otherwise noted, all Python classes, functions, and module variables noted in this document are members of the `khmer` module.

Todo

Use `breathe` to interface with `Doxygen` for better documentation.

9.3 Configuration Objects

9.3.1 C++ API

The declaration of the configuration objects is contained in `lib/khmer_config.hh`.

class `Config`

Config& **`get_active_config`**()

void **`set_active_config`**(Config& *c*)

An *active* configuration object is always present. A reference to this object is supplied via the `get_active_config()` function. The initial settings of the active configuration object are quite conservative. New configuration objects are created with the empty constructor; all settings modifications occur upon already-created instances via their setter methods. The active configuration object can be set via the `set_active_config()` function, which takes a reference to a `Config` object as its only argument.

Except for read-only configuration options, such as extra sanity checking, which are determined at the time of compilation, the configuration options are manipulated via getter/setter methods. The most prominent or useful getter/setter methods are the following:

uint32_t `Config::`**`get_number_of_threads`**`() const`

void `Config::`**`set_number_of_threads`**(uint32_t const *n*)

uint64_t const `Config::`**`get_reads_input_buffer_size`**`() const`

void `Config::`**`set_reads_input_buffer_size`**(uint64_t const *sz*)

9.3.2 Python API

The `Config` objects are exposed in the Python wrapper.

`khmer.`**`get_config`**()

`khmer.`**`set_config`**(*c*)

The C++ getter/setter methods are exposed via the same names in Python.

Todo

The getter/setter methods should be exposed as properties in Python.

9.4 Trace Loggers

Trace loggers can log execution traces and other useful debugging information to files on a per-thread basis. This makes them very useful for debugging multi-threaded code, especially in the absence of advanced commercial debuggers, such as DDT or TotalView. Trace loggers are controlled via several user-configurable variables in the top-level makefile. As of this writing, these variables are `WITH_INTERNAL_TRACING`, `TRACE_STATE_CHANGES`, `TRACE_BUSYWAITS`, `TRACE_SPINLOCKS`, `TRACE_MEMCOPIES`, and `TRACE_DATA`. The `TRACE_` options are ineffective unless `WITH_INTERNAL_TRACING` is set to `true`.

Todo

Replace the editing of makefiles with a configure script or else move to an all-Pythonic regime where the user would edit `setup.cfg`. See issue #9 in the Github issue tracker for the project.

The data pump and read parser code, as well as some of the Bloom filter code, is impregnated with trace loggers. Other parts of the source code could use them as well.

Trace logger objects are not exposed directly via the Python wrapper; they are only available in the C++ API. The trace logger class is declared in the `lib/trace_logger.hh` file.

class `TraceLogger`

Tracing be performed at coarser or finer levels of detail, as desired. An enumeration of named integral constants provides the available levels. The use of `TLVL_ALL` will trace everything which is instrumented for tracing. After that, `TLVL_DEBUG9` is the next finest level of detail. The enumeration ascends to higher and higher numerical values which indicate more coarseness; specifically the ordering is trace levels `TLVL_DEBUG8` through `TLVL_DEBUG0`, followed by `TLVL_INFO9` through `TLVL_INFO0`, and then `TLVL_WARNING`, `TLVL_ERROR`, and `TLVL_CRITICAL`. The special level `TLVL_NONE` means that nothing will be traced even though tracing may be activated at compile time. Note that `TLVL_ALL` corresponds to 0 and `TLVL_NONE` corresponds to 255; this is useful for setting trace levels in method arguments via the Python interface.

Todo

Expose trace level names via the Python interface.

Todo

Allow C++ `TraceLogger` objects to be targets of Python `logging` module?

Two constructors are available for instantiating objects of type `TraceLogger`. One takes the trace level and a `FILE *` stream handle. The other takes the trace level, a file name format string, and a variable number of arguments to `sprintf` into that format string. This form exists so that trace files, named according to logical thread ID, can be created, for example. The trace level argument is the finest requested level of detail which will be traced by the object.

The objects instantiated by these constructors are *function objects* (also sometimes known as *functors* to the chagrin of some mathematicians). This is to say that the objects may be called.

void `TraceLogger::operator()`(uint8_t const *level*, char const* const *format*, ...) `const`

The `level` argument is the desired level of detail. If the object was instantiated for a coarser level than the requested level, then nothing will be logged. The `format` argument is the format string for the underlying `fprintf` call and a variable number of arguments may be supplied for use with this format string.

9.5 Performance Metrics

Performance metrics can be gathered on a per-thread basis and can measure things which are not covered by traditional profiling tools. Such metrics may include the input or output rate in bytes per second, for example. Not all platforms support the high resolution, per-thread timers needed to effectively use these metrics.

The Python wrapper does not presently support reporting on performance metrics.

Todo

Support reporting on performance metrics from within the Python wrapper.

The performance metrics abstract base class is declared in the `lib/perf_metrics.hh` file. This class is subclassed for various specific domains.

class `IPerformanceMetrics`

The class provides a hassle-free stopwatch.

void `IPerformanceMetrics::`**`start_timers`**`()`

void `IPerformanceMetrics::`**`stop_timers`**`()`

These functions record the amount of physical time elapsed since the thread was created and the amount of time that the thread has spent using CPU cores. Two sets of internal scratch variables are used for this purpose: one set of start times and one set of stop times.

Warning: Because of the use of internal scratch variables, these methods are not reentrant. Timer deltas must be collected before new calls can be issued to the stopwatch. This is the trade-off for convenience....

Once start and stop times have been accumulated, then a timer delta can be calculated and stowed in the appropriate category. Categories are determined by keys which are defined in subclasses of the abstract class. The delta accumulator takes a category key as an argument and is declared pure virtual in the abstract base class so that it must be implemented in subclasses where they category keys are enumerated.

void `IPerformanceMetrics::`**`accumulate_timer_deltas`**(uint32_t *metrics_key*)

9.6 Input Data Pumps

An input data pump object copies data from a file into a cache in memory. Since accesses to memory are typically three orders of magnitude faster than to an individual hard disk drive and since many operations to process the data are slower than reading it from a file, it makes sense to stage some of it into memory. Having the data in memory can reduce the latency from accessing it upon demand. And, the cache in memory can be filled more rapidly than it is processed.

The input data pumps are declared in `lib/read_parsers.hh`. All of them derive from an abstract base class.

Todo

Refactor the data pumps into a separate header and implementation file.

---

class `IStreamReader`

Presently, three types of data pumps are implemented.

class `RawStreamReader`

class `GzStreamReader`

class `Bz2StreamReader`

These input data pumps are not exposed via the Python wrapper.

The `IStreamReader` interface defines one method of interest.

uint64_t const **`read_into_cache`**(uint8_t* const *cache*, uint64_t const *cache_size*)

This is a pure virtual method which must be overridden in subclasses. The `cache` parameter receives an argument which is an arbitrary piece of memory which it treats as an array of bytes. The `cache_size` parameter receives an argument which is the size, in bytes, of the cache. The return value is the number of bytes read into the cache from file.

9.6.1 Raw Stream Reader

The raw stream reader is constructed from a file descriptor, such as returned by the `open` system call. An optional read alignment may be supplied to this constructor. Depending on the operating system and file system, this may be used as a chunk size and alignment for *direct I/O*. Otherwise, it is ignored. Direct I/O allows for blocks to copied directly from a block device into user-space memory rather than passing through a kernel-space block cache first. This reduces the number of memory copies involved in processing the data.

Warning: Direct I/O has received some testing within the software, but has not been tested enough to be considered production-ready.

Note: In principle, the file descriptor could number 0 (stdin) and one could create pipelines, but this is not supported at the higher level interfaces.

Reading is currently performed in a synchronous manner, which is fine for most typical use cases of the software since input is not the bottleneck.

Todo

Support asynchronous reading.

9.6.2 Gzip Stream Reader and Decompressor

The stream reader and decompressor for the `gzip` format is based on `zlib`. No direct I/O is supported by this stream reader and its constructor therefore only accepts a file descriptor. Furthermore, data must be copied and decompressed sequentially and cannot be read asynchronously. In the regime that higher level processing is fast, this stream reader is likely to be a bottleneck, especially as there is overhead from decompression. However, as pipelining support does not yet exist in the software, providing native support for a popular compression format makes sense. Also, some users of the software may not be familiar with standard Unix compression tools, such as `gzip`; built-in support of popular compression formats removes a barrier to entry for these users.

Todo

Implement higher level support for pipelining so that parallelized decompressors can feed a raw stream reader, assuming that they can output decompressed data to stdout and do so in order. Alternatively, if a parallelized variant of `zlib` can be found, then that should be used in place of `zlib` for native support.

9.6.3 Bzip2 Stream Reader and Decompressor

The stream reader and decompressor for the `bzip2` format is based on the `bzip2` library. The same notes and considerations for the `gzip` stream reader also apply to this one as well.

As a historical note, it is worth mentioning that the logic for reading from a `bzip2`-compressed file stream is significantly more complicated than for raw or `gzip`-compressed streams because of the way the library API is structured and the nature of the compression format. Prior to the architecture being described, data pumps and reads parsers were tightly coupled and implementing a `bzip2` data pump in that architecture would have been very painful. As it turns out, the current architecture preemptively fixed a bug in the old `gzip` data pump before it was reported against that architecture. So, this decoupled design has already paid for itself several times over.

9.7 Thread Identity Maps

Higher level processing requires that threads be able to persistently work with the same set of data. A thread does not inherently "know" what its index into a particular lookup table is. However, this index can be mapped to an OS-native identifier for a thread. Using an appropriate system call, a thread can query its own native identifier from the operating system and then use this as a map key to find its logical identifier within the software. This logical identifier serves as the thread's index into any lookup tables which it may need to use.

The self-identification is also important on the grounds of a software engineering principle: don't break existing interfaces. Prior to the current architecture, the code was not thread-safe. In order to add thread-safety in a reliable manner and not break existing interfaces, self-identification of threads was necessary.

The thread identity map class is declared in the `lib/thread_id_map.hh` file.

class `ThreadIDMap`

This class is not exposed via the Python wrapper as it is an internal mechanism. And, the implementation of the class varies according to operating system. The only important method for those who wish to avail themselves to this bookkeeping method is the one which returns the logical identifier (lookup table index) of the current thread.

uint32_t const `ThreadIDMap::get_thread_id()`

New entries are added to the map as new threads call this method for the first time. Thus, the bookkeeping is automatic and does not get in the way of the developer.

9.8 Cache Managers

A cache manager provides memory into which an input data pump may copy. The provided memory is segmented on a per-thread basis. On machines with multiple NUMA nodes, this can help performance by decreasing the likelihood of cross-node fetches and stores. More importantly, it provides an association between a particular thread and a particular cache segment, so that higher level processing, such as parsing, can always be guaranteed to operate on the same contiguous portion of memory.

Todo

Implement pinning of threads to specific cores on operating systems which support this. Preventing the migration of threads between cores should mostly eliminate cross-node fetches and stores.

The `lib/read_parsers.hh` file declares the cache manager and cache manager segment classes.

class `CacheManager`

class `CacheManagerSegment`

As multiple threads share access to the same data pump, the cache manager orchestrates access to this resource. Internally, a spinlock is used to limit access to one thread at a time.

Todo

Increase period of spinlock trials from once per iteration to something greater, similar to what the other busywaiters which perform atomic tests use.

Internally, a `ThreadIDMap` is used to match a current thread with its corresponding entry in the table of cache segments. A convenience method is provided for the current thread to find its corresponding cache segment, creating it if it doesn't already exist.

CacheSegment& `CacheManger::_get_segment` (bool const *higher=false*)

This is a private method used only within cache mangers. The `higher` parameter is vestigial remnant from an earlier implementation and can likely be removed.

Todo

Remove the `higher` parameter from `_get_segment()`.

Developers wishing to use a cache manager rather than muck around in its implementation will probably find the following methods most useful.

bool const `CacheManager::`**`has_more_data`**`()`

uint64_t const `CacheManager::`**`get_bytes`**`(`uint8_t* const *buffer*, uint64_t *buffer_len*`)`

void `CacheManager::`**`split_at`**`(`uint64_t const *pos*`)`

The `has_more_data()` method queries both the underlying stream and the current cache segment to see if more data is available. If both the underlying stream is exhausted and the memory cursor, which tracks how much of a cache segment has been accessed since its last refill, is at the end of the segment, then no more data is considered to be available and the current thread hits a synchronization barrier to wait for the other threads to finish.

The `get_bytes()` method copies up to `buffer_len` bytes of memory from the current cache segment into the supplied buffer `buffer`. All bookkeeping, such as replenishing the cache segment from the underlying stream, is handled behind the scenes. The method also copies memory from the appropriate *copyaside buffer* as necessary. Copyaside buffers are created by the `split_at()` method and represent extensions to the current cache segment.

Todo

Expose the underlying memory segments directly to higher level processing, such as parsing, to eliminate the memory copy overhead that `get_bytes()` carries. Note that this comes at the cost of some horrid bookkeeping on the part of the higher level functions. The `get_bytes()` method exists to handle this bookkeeping.

The `split_at()` method copies up to `pos` bytes from the beginning of the current cache segment into a copyaside buffer. The copyaside buffer will then be available for the previous (in terms of lookup table index modulo the number of threads) cache segment. This method helps with multi-threaded parsing of files when parser offsets into a file do not correspond with record boundaries. A parser can scan forward to the next record boundary and then set the scanned over bytes aside to be appended to the cache segment which contains the beginning of the partial record.

The initial implementation of the cache manager used *setaside buffers*, which were just reserved portions of cache segments and no memory copies were performed. However, the bookkeeping was quite complicated and after several bugs slipped through the cracks, the setaside buffer logic was converted to copyaside buffers. The cost of the memory copies is essentially nothing in the typical use cases encountered by the software. Copyaside buffers are also much more amenable to asynchronous refilling of cache segments, should that be supported at a later point.

Todo

Implement asynchronous refills of cache segments.

9.9 Reads and Read Pairs

Reads are simple data structures which contain genomic sequences, as well identifiers and quality scores for those sequences. The class is declared in `lib/read_parsers.hh`.

class `Read`

The Python wrapper exposes an interface to reads.

class khmer.**Read**

The data members are accessed as properties. These mimic the access keys for screed records.

Read.**name**

Read.**sequence**

Read.**accuracy**

Read.**annotations**

No distinction is currently made between FASTA and FASTQ reads.

Todo

Create an IRead abstract base class and subclass for FASTA and FASTQ record types. This would remove wasted fields for FASTA records and allow the type of records being used at any level of processing.

Read pairs are two reads bound together in a STL pair. This is intended to track sequences with paired ends.

9.10 Read Parsers

9.10.1 C++ API

Read parsers create the aforementioned Read objects. The lib/read_parsers.hh file declares an abstract base class as well as FASTA and FASTQ parsers derived from that. These are made available from within a namespace which encapsulates most classes in the lib/read_parsers.hh file.

class IParser

class FastaParser

class FastqParser

An instance of the appropriate subclass is created via a factory method provided by the abstract class. This method infers the correct subclass instance to create based on file name extension. The file name is required but the other arguments are optional. If the other arguments are supplied, then they override the defaults from the active Config object.

IParser* const IParser::**get_parser**(std::string const& *ifile_name*, uint32_t const *number_of_threads*, uint64_t const *cache_size*, uint8_t const *trace_level*)

Todo

Sniff file type rather than rely on extension.

Just as the CacheManager maintains per-thread state in CacheSegment objects, the parser classes maintain per-thread state in special objects as well.

class ParserState

The parser state maintains a line buffer, among other things, and tracks how much of it has been parsed by each call to the parser.

The IParser interface provides some useful methods.

bool IParser::**is_complete**()

void IParser::**imprint_next_read**(Read& *the_read*)

void IParser::**imprint_next_read_pair**(ReadPair& *the_read_pair*, uint8_t *mode=PAIR_MODE_ERROR_ON_UNPAIRED*)

The is_complete() method checks if parsing of the current stream is complete and blocks in a synchronization barrier if it is but some threads are still working.

The imprint_next_read() method attempts to parse another read from the file stream and create a Read object from it. Note that a legacy method get_next_read is still available but its use in new code is discouraged. The legacy method involves an additional memory copy.

The imprint_next_read_pair() method attempts to parse a pair of reads from the file stream a create a *ReadPair* object from them. Currently, this has two implemented modes of operation with a third one planned. The modes are PAIR_MODE_ALLOW_UNPAIRED, PAIR_MODE_IGNORE_UNPAIRED, and PAIR_MODE_ERROR_ON_UNPAIRED. The first one is not yet implemented; it may be useful for filtering or diverting paired or unpaired reads out of a stream. The PAIR_MODE_IGNORE_UNPAIRED mode simply ignores unpaired reads and only returns paired reads. The PAIR_MODE_ERROR_ON_UNPAIRED mode raises an exception if an unpaired read is encountered. As a note, both the old-style ("/1" and "/2") and new-style ("1..." and "2:...") Illumina read pairs are detected from sequence identifiers.

Todo

Implement PAIR_MODE_ALLOW_UNPAIRED mode.

Todo

Place burden of input parsing and output formatting on Read obects rather than on parser methods. Demote parsers to role of facilitator. Maybe?

9.10.2 Python Wrapper

The Python wrapper exposes a read parser class.

class khmer.**ReadParser**

This class has no subclasses, but handles various formats appropriately. An instance of the class is an iterator, which produces one read at a time. There is also a method for iterating over read pairs and the class exposes the same constants for controlling its behavior as the underlying C++ class does.

ReadParser.**iter_read_pairs**(*pair_mode*)

9.11 k-mer Counters and Bloom Filters

9.11.1 C++ API

The Bloom filter counting is described elsewhere and so we won't go into details of it here. Some of the methods of the hash tables has been granted thread safety and can use the thread-safe IParser objects.

class Hashtable

class Hashbits

void Hashtable::**consume_fasta**(IParser* *parser*, unsigned int& *total_reads*, unsigned long long& *n_consumed*, HashIntoType *lower_bound*, HashIntoType *upper_bound*, CallbackFn *callback*, void* *callback_data*)

void Hashbits::**consume_fasta_and_tag**(IParser* *parser*, unsigned int& *total_reads*, unsigned long long& *n_consumed*, CallbackFn *callback*, void* *callback_data*)

For legacy support, methods with signatures that have a file name parameter rather than a IParser parameter are still provided as well. (They wrap the ones with the parser parameter.)

As with the cache managers and read parsers, the hashtables track per-thread state.

class Hasher

Since more than one pool of threads (e.g., one set of threads per reads parser and one reads parser per file stream) may be used with a particular hash table object, the hash table objects internally maintain the notion of thread pools. The universally unique identifier (UUID) of an object (e.g., a reads parser) is used to map to the correct thread pool. This is behind-the-scenes accounting and a developer should generally not have to worry about this. But, if you are converting another method to be thread-safe and it can take different reads parsers on different invocations, then be sure to consider this.

Todo

Drop more logic currently implemented in Python to C++ to gain multi-threading efficiencies. Not everything can really scale well using the existing interfaces working in Python.

Todo

Cache k-mers to hash in small buckets which correspond to regions of the hash tables. This will allow for multiple updates per memory page and reduce the number of CPU cache misses.

Todo

Abstact the counter storage from the hash functions. A number of open issues can be addressed by doing this. The counter storage might be better implemented with partial template specialization than with subclassing. For small hash tables sizes, not hashing makes more sense because every possible k-mer in the k-mer space can be addressed directly in memory. Counter storage will be most efficient for powers-of-two numbers of bits per counter. Blah, blah... these and other thoughts are discussed more thoroughly in the various GitHub issues involving them.

9.11.2 Python Wrapper

The hash table objects have methods which take ReadParser objects and invoke the appropriate C++ methods underneath the hood.

new_hashtable.**consume_fasta_with_reads_parser**(*rparser*)

new_counting_hash.**consume_fasta_and_tag_with_reads_parser**(*rparser*)

Todo

Convert factory functions into callable classes and properly attribute those classes.

9.12 Python Wrapper

The Python wrapper resides in python/_khmermodule.cc. C++ code is used to call the CPython API to bind some of the C++ classes and methods to Python classes and methods. Some of the newer additions to the wrapper, such as the Read and ReadParser classes should be considered models for future additions as they expose callable

classes with properties and iterators and which look just like Python classes for the most part. Much of the older code relies on factory functions to create objects and those objects are not very Pythonic. The newer additions are also much less cluttered and more readable (though the author of this sentence may be biased in this regard).

Todo

Use SWIG to generate the interface. Maybe?

CHAPTER 10

Miscellaneous implementation details

Partition IDs are "stored" in FASTA files as an integer in the last tab-separated field. Yeah, dumb, huh?

CHAPTER 11

Development miscellany

11.1 Third-party use

We ask that third parties who build upon the codebase to do so from a versioned release. This will help them determine when bug fixes apply and generally make it easier to collaborate. If more intensive modifications happen then we request that the repository is forked, again preferably from a version tag.

11.2 Build framework

'make' should build everything, including tests and "development" code.

11.3 Coding standards

All plain-text files should have line widths of 80 characters or less unless that is not supported for the particular file format.

For C++, we use Todd Hoff's coding standard, and astyle -A10 / "One True Brace Style" indentation and bracing. Note: @CTB needs emacs settings that work for this.

Vim users may want to set the ARTISTIC_STYLE_OPTIONS shell variable to "-A10 –max-code-length=80" and run `:%!astyle` to reformat. The four space indentation can be set with:

```
set expandtab
set shiftwidth=4
set softtabstop=4
```

For Python, PEP 8 is our standard. The `pep8` and `autopep8` Makefile targets are helpful.

Code, scripts, and documentation must have its spelling checked. Vim users can run:

```
:setlocal spell spelllang=en_us
```

Use *]s* and *[s* to navigate between misspellings and *z=* to suggest a correctly spelled word. *zg* will add a word as a good word.

GNU's *aspell* can also be used to check the spelling in a single file:

```
aspell check --mode ccpp $filename
```

11.4 Code Review

Please read 11 Best Practices for Peer Code Review.

See also Code reviews: the lab meeting for code and the PyCogent coding guidelines.

11.5 Checklist

Copy and paste the following into a pull-request when it is ready for review:

```
- [ ] Is it mergable
- [ ] Did it pass the tests?
- [ ] If it introduces new functionality in scripts/ is it tested?
  Check for code coverage.
- [ ] Is it well formatted? Look at `pep8`/`pylint`, `cppcheck`, and
  `make doc` output. Use `autopep8` and `astyle -A10 --max-code-length=80`
  if needed.
- [ ] Is it documented in the Changelog?
- [ ] Was spellcheck run on the source code and documentation after changes
  were made?
```

11.6 git and github strategies

Still in the works, but read this.

Make a branch on ged-lab (preferred so others can contribute) or fork the repository and make a branch there.

Each piece or fix you are working on should have its own branch; make a pull- request to ged-lab/master to aid in code review, testing, and feedback.

If you want your code integrated then it needs to be mergeable

Example pull request update using the command line:

1. **Clone the source of the pull request (if needed)** `git clone git@github.com:mr-c/khmer.git`
2. **Checkout the source branch of the pull request** `git checkout my-pull-request`
3. **Pull in the destination of the pull request and resolve any conflicts** `git pull git@github.com:ged-lab/khmer.git master`
4. Push your update to the source of the pull request `git push`
5. Jenkins will automatically attempt to build and test your pull requests.

11.7 Testing

`./setup.py nosetest` is the canonical way to run the tests. This is what `make test` does.

11.8 Code coverage

Jenkins calculates code coverage for every build. Navigate to the results from the master node first to view the coverage information.

Code coverage should never go down and new functionality needs to be tested.

11.9 Pipelines

All khmer scripts used by a published recommended analysis pipeline must be included in scripts/ and meet the standards therein implied.

11.10 Command line scripts

Python command-line scripts should use '-' instead of '_' in the name. (Only filenames containing code for import imported should use _.)

Please follow the command-line conventions used under scripts/. This includes most especially standardization of '-x' to be hash table size, '-N' to be number of hash tables, and '-k' to always refer to the k-mer size.

Command line thoughts:

> If a filename is required, typically UNIX commands don't use a flag to specify it.
>
> Also, positional arguments typically aren't used with multiple files.
>
> CTB's overall philosophy is that new files, with new names, should be created as the result of filtering etc.; this allows easy chaining of commands. We're thinking about how best to allow override of this, e.g.
>
> ```
> filter-abund.py <kh file> <filename> [ -o <filename.keep> ]
> ```

All code in scripts/ must have automated tests; see tests/test_scripts.py. Otherwise it belongs in sandbox/.

When files are overwritten, they should only be opened to be overwritten after the input files have been shown to exist. That prevents stupid command like mistakes from trashing important files.

It would be nice to allow piping from one command to another where possible. But this seems complicated.

CTB: should we squash output files (overwrite them if they exist), or not? So far, leaning towards 'not', as that way no one is surprised and loses their data.

A general error should be signaled by exit code *1* and success by *0*. Linux supports exit codes from *0* to *255* where the value *1* means a general error. An exit code of *-1* will get converted to *255*.

CLI reading:

> http://stackoverflow.com/questions/1183876/what-are-the-best-practices-for-implementing-a-cli-tool-in-perl
>
> http://catb.org/esr/writings/taoup/html/ch11s06.html
>
> http://figshare.com/articles/tutorial_pdf/643388

11.11 Python / C integration

The Python extension that wraps the C++ core of khmer lives in khmer/_khmermodule.CC

This wrapper code is tedious and annoying so we use a static analysis tool to check for correctness.

https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html

Developers using Ubuntu Precise will want to install the gcc-4.6-plugin-dev package

Example usage:

```
CC="/home/mcrusoe/src/gcc-plugin-python/gcc-python-plugin/gcc-with-cpychecker
--maxtrans=512" python setup.py build_ext 2>&1 | less
```

False positives abound: ignore errors about the C++ standard library. This tool is primarily useful for reference count checking, error-handling checking, and format string checking.

Errors to ignore: “Unhandled Python exception raised calling ‘execute’ method”, “AttributeError: ‘NoneType’ object has no attribute ‘file”’

Warnings to address:

```
khmer/_khmermodule.cc:3109:1: note: this function is too complicated
for the reference-count checker to fully analyze: not all paths were
analyzed
```

Adjust –maxtrans and re-run.

```
khmer/_khmermodule.cc:2191:61: warning: Mismatching type in call to
Py_BuildValue with format code "i" [enabled by default]
  argument 2 ("D.68937") had type
    "long long unsigned int"
  but was expecting
    "int"
  for format code "i"
```

See below for a format string cheatsheet One also benefits by matching C type with the function signature used later.

“I” for unsigned int “K” for unsigned long long a.k.a khmer::HashIntoType.

CHAPTER 12

Deploying the khmer project tools on Galaxy

We are developing the support for running normalize-by-median in Galaxy.

When this is mature we will make a Galaxy Tool Shed version available for easier installation.

12.1 Install the tools & tool description

If your installation uses a virtualenv be sure to activate it in your terminal before continuing.

```
pip install --no-clean khmer
```

Move to the *tools* directory in your Galaxy installation and copy in the tool definition file.:

```
cd tools
mkdir khmer
ln -s build/khmer/scripts/normalize-by-median.xml .
```

Add the following to your tool_conf.xml inside the <toolbox> tag:

```
<section id="khmer-protocols-extra" name="khmer protocols">
<tool file="khmer/normalize-by-median.xml" />
</section>
```

Then (re)start Galaxy.

12.2 Single Output Usage

For one or more files into a single file:

#. Choose 'Normalize By Median' from the 'khmer protocols' section of the 'Tools' menu.

#. Compatible files already uploaded to your Galaxy instance should be listed. If not then you may need to set their datatype manually.

#. After selecting the input files specify if they are paired-interleaved or not.

#. Specify the sample type or show the advanced parameters to set the tablesize yourself. Consult *Choosing table sizes for khmer* for assistance.

CHAPTER 13

Known Issues

Some users have reported that normalize-by-median.py will utilize more memory than it was configured for. This is being investigated in https://github.com/ged-lab/khmer/issues/266

Some FASTQ files confuse our parser when running with more than one thread. For example, while using load-into-counting.py. If you experience this then add "–threads=1" to your command line. This issue is being tracked in https://github.com/ged-lab/khmer/issues/249

If your counting or presence table gets truncated, perhaps from a full filesystem, then our tools currently will get stuck. This is being tracked in https://github.com/ged-lab/khmer/issues/247 and https://github.com/ged-lab/khmer/issues/96 and https://github.com/ged-lab/khmer/issues/246

Paired-end reads from Casava 1.8 currently require renaming for use in normalize-by-median and abund-filter when used in paired mode. The integration of a fix for this is being tracked in https://github.com/ged-lab/khmer/issues/23

annotate-partitions.py only outputs FASTA even if given a FASTQ file. This issue is being tracked in https://github.com/ged-lab/khmer/issues/46

A user reported that abundance-dist-single.py fails with small files and many threads. This issue is being tracked in https://github.com/ged-lab/khmer/issues/75

CHAPTER 14

How to make a khmer release

Michael R. Crusoe is the current release maker. This is his checklist.

1. The below should be done in a clean checkout:

   ```
   cd `mktemp -d`
   git clone git@github.com:ged-lab/khmer.git
   cd khmer
   ```

2. (Optional) Check for updates to versioneer:

   ```
   pip install versioneer
   versioneer-installer

   git diff

   ./setup.py versioneer
   git diff
   git commit -m -a "new version of versioneer.py"
   # or
   git checkout -- versioneer.py khmer/_version.py khmer/__init__.py MANIFEST.in
   ```

3. Review the git logs since the last release and diffs (if needed) and ensure that the `Changelog` is up to date:

   ```
   git log --minimal --patch `git describe --tags --always --abbrev=0`..HEAD
   ```

4. Review the issue list for any new bugs that will not be fixed in this release. Add them to `doc/known-issues.txt`

5. Verify that the build is clean: http://ci.ged.msu.edu/job/khmer-multi/

6. Tag the branch with the release candidate version number prefixed by the letter 'v':

   ```
   new_version=1.0.1
   git tag v${new_version}-rc1
   git push --tags git@github.com:ged-lab/khmer.git
   ```

7. Test the release candidate. Bonus: repeat on Mac OS X:

   ```
   cd ..
   virtualenv testenv1
   virtualenv testenv2
   virtualenv testenv3
   virtualenv testenv4
   # First we test the tag
   ```

```
cd testenv1
source bin/activate
git clone --depth 1 --branch v${new_version}-rc1 https://github.com/ged-lab/khmer.git
cd khmer
make install
make test
normalize-by-median.py --version # double-check version number

# Secondly we test via pip

cd ../../testenv2
source bin/activate
pip install -e git+https://github.com/ged-lab/khmer.git@v${new_version}-rc1#egg=khmer
cd src/khmer
make dist
make install
make test
normalize-by-median.py --version # double-check version number
cp dist/khmer*tar.gz ../../../testenv3/

# Is the distribution in testenv2 complete enough to build another
# functional distribution?

cd ../../../testenv3/
source bin/activate
pip install khmer*tar.gz
tar xzf khmer*tar.gz
cd khmer*
make dist
make test
```

8. Publish the new release on the testing PyPI server:

   ```
   python setup.py register --repository test
   ```

   Change your PyPI credentials as documented in https://wiki.python.org/moin/TestPyPI:

   ```
   python setup.py sdist upload -r test
   ```

   Test the PyPI release in a new virtualenv:

   ```
   cd ../../testenv4
   source bin/activate
   pip install screed
   pip install -i https://testpypi.python.org/pypi --pre --no-clean khmer
   normalize-by-median.py --version 2>&1 | awk ' { print $2 } '
   cd build/khmer
   make test
   ```

9. Create the final tag and publish the new release on PyPI (requires an authorized account).:

   ```
   cd ../../../khmer
   git tag v${new_version}
   python setup.py register sdist upload
   ```

10. Delete the release candidate tag and push the tag updates to github.:

    ```
    git tag -d v${new_version}-rc1
    git push git@github.com:ged-lab/khmer.git
    git push --tags git@github.com:ged-lab/khmer.git
    ```

11. Make a binary wheel on OS X.:

```
virtualenv build
cd build
source bin/activate
pip install khmer==${new_version}
pip install wheel
cd build/khmer
./setup.py bdist_wheel upload
```

12. Tweet about the new release. Optionally send email including the contents of the ChangeLog to khmer@lists.idyll.org and khmer-announce@lists.idyll.org

14.1 Upstream sources

ez_setup.py is from https://bitbucket.org/pypa/setuptools/raw/bootstrap/

versioneer.py is from https://raw.github.com/warner/python-versioneer/master/versioneer.py

Before major releases they should be examined to see if there are new versions available and if the change would be useful

14.2 Explanation

Versioneer, from https://github.com/warner/python-versioneer, is used to determine the version number and is called by Setuptools and Sphinx. See the files `versioneer.py`, the top of `khmer/__init__.py`, `khmer/_version.py`, `setup.py`, and `doc/conf.py` for the implementation.

The version number is determined through several methods: see https://github.com/warner/python-versioneer#version-identifiers

If the source tree is from a git checkout then the version number is derived by `git describe --tags --dirty --always`. This will be in the format `${tagVersion}-${commits_ahead}-${revision_id}-${isDirty}`. Example: `v0.6.1-18-g8a9e430-dirty`

If from an unpacked tarball then the name of the directory is queried.

Lacking either of the two git-archive will record the version number at the top of `khmer/_version.py` via the `$Format:%d$` and `$Format:%H$` placeholders enabled by the "export-subst" entry in `.gitattributes`.

Non source distributions will have a customized `khmer/_version.py` that contains hard-coded version strings. (see `build/*/khmer/_version.py` after a `python setup.py build` for an example)

`ez_setup.py` bootstraps Setuptools (if needed) by downloading and installing an appropriate version

CHAPTER 15

Crazy ideas

1. A JavaScript preprocessor to do things like count k-mers (HLL), and do diginorm on data as uploaded to server. Inspired by a paper that Titus reviewed for PLoS One; not yet published.

CHAPTER 16

Contributors and Acknowledgements

khmer is a product of the GED lab at Michigan State University,

http://ged.msu.edu/

—

C. Titus Brown <ctb@msu.edu> wrote the initial ktable and hashtable implementations, as well as hashbits and counting_hash.

Jason Pell implemented many of the C++ k-mer filtering functions.

Qingpeng contributed code to do unique k-mer counting.

Adina Howe, Rosangela Canino-Koning, and Arend Hintze contributed significantly to discussions of approaches and algorithms; Adina wrote a number of scripts.

Jared T. Simpson (University of Cambridge, Sanger Institute) contributed paired-end support for digital normalization.

Eric McDonald thoroughly revised many aspects of the code base, made much of the codebase thread safe, and otherwise improved performance dramatically.

Michael R. Crusoe is the new maintainer of khmer.

MRC 2014-05-07

CHAPTER 17

An incomplete bibliography of papers using khmer

17.1 Digital normalization

Multiple Single-Cell Genomes Provide Insight into Functions of Uncultured Deltaproteobacteria in the Human Oral Cavity. Campbell et al., PLoS One, 2013, doi:10.1371/journal.pone.0059361. [paper link]

Insights into archaeal evolution and symbiosis from the genomes of a nanoarchaeon and its inferred crenarchaeal host from Obsidian Pool, Yellowstone National Park. Podar et al., Biology Direct, 2013 doi:10.1186/1745-6150-8-9. [paper link]

CHAPTER 18

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Michigan State University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

CHAPTER 19

Indices and tables

- *genindex*
- *modindex*
- *search*

k