

Meraculous-2D Assembler Manual

Lawrence Berkeley National Lab
Genomics Division
DOE Joint Genome Institute
Walnut Creek, CA

Revision 2.2.4

Contents

1. [Before you begin](#)
 - 1.1. [Capabilities and limitations](#)
 - 1.2. [Operating system requirements](#)
 - 1.3. [Hardware considerations](#)
 - 1.4. [Internal job execution and control](#)
 - 1.5. [Availability and Getting help](#)
2. [Installation](#)
 - 2.1. [Software dependencies](#)
 - 2.2. [Installation procedure](#)
 - 2.3. [Test run](#)
3. [Running Meraculous](#)
 - 3.1. [Workflow steps and stages](#)
 - 3.2. [Input data preparation](#)
 - 3.3. [Run configuration](#)
 - 3.4. [Executing the run](#)
 - 3.5. [Working in stages](#)
4. [Evaluating output and Troubleshooting](#)
 - 4.1. [Key output files](#)
 - 4.2. [Logs and error handling](#)
 - 4.3. [Run evaluation script](#)
 - 4.4. [Troubleshooting tips](#)
5. [Technical Notes](#)
 - 5.1. [Diploid assembly](#)
6. [Citing and feedback](#)

Meraculous-2D (referred to from here on as simply Meraculous) is a whole genome assembler for Next Generation Sequencing data geared to eukaryotic genomes. It is a hybrid k-mer/read-based assembler that capitalizes on the high accuracy of Illumina sequence by eschewing an explicit error correction step which we argue to be redundant with the assembly process. Meraculous achieves high performance with large datasets by utilizing lightweight data structures and multi-threaded parallelization, allowing assembling human-sized genomes on commodity clusters in reasonable time. The process pipeline implements a highly transparent and portable model of job control and

monitoring where different assembly stages can be executed and re-executed separately or in unison on a wide variety of architectures

1. BEFORE YOU BEGIN

1.1 Capabilities and limitations

- Currently Meraculous works with Illumina data only. It relies on Illumina naming conventions and Phred-like sequence quality scores. Long-read/low-depth sequencing platforms are not supported at this time.
- An overall mean depth of read coverage of at least 30x is strongly recommended. Low-coverage datasets will likely result in a highly fragmented assembly or an aborted process altogether.
- Meraculous doesn't perform any explicit coverage optimization (e.g. down-sampling) of input reads. In most circumstances, excessive coverage will not affect assembly quality, but in large assemblies it can lead to longer run times and potential memory overruns. Therefore, we recommend keeping the average depth of read coverage below 100x.
- If your input sequence data has been pre-trimmed or filtered, please make sure that all reads still have their pairs and that the minimum length doesn't fall below the kmer size you've chosen for this dataset.
- Although it is capable of assembling small bacterial genomes, this assembler may not be the most efficient choice for bacterial projects.
- Meraculous relies heavily on distributed and threaded computing and will perform best on a multi-core server or on a cluster with high bandwidth inter-node communication. For more on this, see sections ['Operating System requirements'](#) and ['Hardware considerations'](#)

Meraculous performs the assembly by first traversing a subgraph of the k-mer (deBruijn) graph of oligonucleotides with unique high quality extensions and building a set of high-confidence contigs (called UUtigs from here on) where each k-mer is represented only once and no further unique extension is possible. If running in diploid mode, Meraculous identifies "bubbles" in the contig graph and merges qualifying bubble-contigs into longer contiguous units we call "diplotigs" (i.e., they contain diploid/heterozygous sequence). Then, the order and orientation (a.k.a. "ono") of UUtigs or diplotigs is determined and gaps are closed using linkage information derived from mapping paired reads back to the UUtigs (or diplotigs).

While there are numerous factors that can affect the accuracy and performance of this process, the following are of utmost importance:

Depth of coverage distribution

Meraculous relies on the coverage profile at several key points in the process. Optimal assembly is possible only when the depth profile of the genome can be clearly distinguished from any low-frequency noise or contamination. The depth distribution is also used to distinguish haplotype variants from homozygous regions and repeats during various assembly stages.

Library composition

For increased contiguity and layout correctness, sequence data from a long-range paired-end or mate-pair (also known as 'jumping') library should be included, with its mean insert size well over the size of the largest repetitive elements in the genome. Since repeat composition can often be complex and unknown ahead of time, it is recommended that a balanced mix of several libraries of different insert sizes is used. Artifacts in such libraries (chimeras, insert-less pairs, untrimmed adapter, etc.) or non-Poisson insert size distribution can result in highly fragmented assembly. However, there are options a user can set to mitigate those effects, and it is crucial that those options are set correctly. See section '[Run configuration](#)' for more details.

Ploidy

Polymorphisms in diploid genomes lead to additional ambiguities and conflicts in the assembly graph traversal and need to be handled appropriately by selecting one of two diploid assembly modes. The choice largely depends on the expected polymorphism rate relative to the chosen k-mer size k . Genomes with relatively infrequent variants (rate $< 1/k$) are best assembled by identifying "bubbles" in the assembly graph and selecting a single path through each bubble and taking the alternative

paths out of the picture (until they can be re-introduced for haplotype phasing step much later in the process). This is accomplished by selecting *diploid mode 1*.

On the other hand, a genome that's highly polymorphic (rate $> 1/k$) is best assembled in a way that keeps both haplotypes intact. For this, *diploid mode 2* must be switched on. It often makes sense to try both modes and then evaluate the output to see which one is better suited for the data set. For more on this see section ['Diploid Assembly'](#).

Repeat content

Like most de-novo assemblers, Meraculous attempts to resolve ambiguities caused by repeats using paired-end linkage and distance information. Because repeating elements can take complex shapes and/or can be confounded by variants in diploid genomes, many repeats may remain unresolved as Meraculous will not make ambiguous joins. Identical repeat copies may "collapse" into what then looks like a unique contig that "falls out" of the scaffold, thus adding to the total number of scaffolds and contigs. During gap closure, some of these repeats will be recovered if a repeat-induced gap can be fully traversed by "kmer-walking".

Read data quality

Meraculous avoids an explicit error correction step instead relying on k-mer coverage and base quality scores (e.g. low quality extensions of k-mers are ignored during the graph building stage). K-mers containing sequencing errors are expected to occur at a much lower frequency than true genomic k-mers and can therefore be eliminated using the depth cutoff. High errors rates in the sequence, however, can still hinder the assembly (especially in diploid mode) and/or increase the memory requirements since a greater number of unique k-mers would result from sequencing error.

k-mer size

If the chosen k-mer size is too short for the genome then there will be too many non-unique k-mers. If it's too long then any given k-mer will be more likely to include a sequencing error. Both situations hinder assembly, so it's important to arrive at a "sweet spot" in the middle. There are 3rd party tools available for auto-determining the optimal k-mer size from actual sequencing data (e.g. [kmergenie](#)) . Alternatively, we advise to start with 3-4 pilot runs with different k-mer sizes, stopping after the *meraculous_mercount* stage and reviewing the k-mer frequency histogram (*mercount.png*, see ['Key output files'](#) for an example). As a rule of thumb, aim for the largest *k* that yields the main haploid peak at least $\sim 30X$, and a distinct trough to the left of it that's at most $1/10$ of the peak height.

Insert size distribution and library bias

Meraculous uses user-defined estimated average insert size and standard deviation values to set various cutoffs during the calculation of the actual assembly-based insert size distribution. Therefore, it's important that these estimates are close to reality.

1.2 Operating system requirements

Meraculous can run on any 64-bit Linux system. This release was developed and tested on Debian 6.0.7 Linux, kernel 2.6.32-5-amd64.

Meraculous can execute distributed/parallel jobs on either a single multi-core system or on a cluster. In the latter case, the cluster array job submission is executed via a wrapper script `cluster_submit.sh`. Depending on your site's cluster configuration, this script may need to be modified by your sysadmin to ensure proper functionality, or even substituted by your own wrapper. This release was tested with a Linux cluster running Univa Grid Engine 8.1.4. At the minimum, the current implementation requires that your cluster scheduler accepts `qsub`, `qstat`, and `qacct` commands for submitting and monitoring of the jobs.

1.3 Hardware considerations

Disk space and memory requirements depend on many factors, but the following guidelines are a good estimate.

Disk space

This will differ dramatically depending on the cleanup policy chosen. For details on the different cleanup options see section ['Executing the run'](#).

Total input sequence (Gbp)	cleanup_level 0	cleanup_level 1	cleanup_level 2
450 (Human)	6.9 TB	2.9 TB	39 GB
13 (N.sebacea)	67 GB	49 GB	200 MB
1.2 (E.coli)	13 GB	7.6 GB	42 MB

Memory

Where possible, memory usage is optimized by partitioning the input data into blocks that can fit under smaller memory caps. However, for larger genomes like H.sapiens, the memory-limiting step is often UUtig generation where all the building blocks must be kept in memory at once.

Genome Size (Gbp)	Total sequence earmarked for contiging (Gbp)	Memory required (GB)
3.1 (H.sapiens)	290	109
0.7 (S.bicolor)	63	18
0.05 (N.fluitans)	10	3.7
0.005 (E.coli)	0.7	0.2

Multi-threading

Many Meraculous components run as threads on multiple cores of a single computer, while others are single-threaded. To take advantage of multi-threading users must specify how many cores are available (parameters `local_num_procs` and `cluster_slots_per_task`). Keep in mind that while multi-threading greatly improves run times it has no effect on the memory usage since the size of the input data chunk is independent of the threading level chosen.

1.4 Internal job execution and control

When following the progress of the run or reviewing the log files, it is important to understand how the Meraculous components get executed on the system. Meraculous makes heavy use of distributed task arrays for parallel execution. First, the data is sub-divided into chunks of smaller size. Then a "command set" containing all the chunks is submitted for execution either locally or on the cluster as task arrays. Meraculous waits for the local system or the cluster scheduler software to signal that all the parallel tasks have been completed successfully before continuing. In the event of failure, users can opt to automatically retry the failed tasks. If there are still failed tasks the program terminates. After examining the error logs and making corrections to the data or the settings, user can resume the assembly from the point where it was terminated (see section '[Working in Stages](#)'). While this will re-form the command set from scratch, only those tasks that haven't successfully completed will actually get re-executed.

1.5 Availability and getting help

Meraculous source code along with a small test dataset is available from <http://sourceforge.net/projects/meraculous20>.

Questions can be addressed to Eugene Goltsman at egoltsman@lbl.gov
If you're running into problems with installation, please include the capture of the screen output of install.sh. If there are errors during Meraculous execution that you don't understand, please send your config file and the log/meraculous.log file from your run.

2. INSTALLATION

2.1 Software dependencies

At the time of the installation the following software should be installed on your system:

cmake >= 2.8

GCC g++ >= 4.8

libgd >= 2.0

GNU make 3.81

Boost C++ library >= 1.57.0

Perl (>= 5.10)

[Log4perl.pm](#) (>= 1.31)

gnuplot (>= 3.7)

qqacct (optional but highly recommended for Grid Engine cluster environments:
<http://portal.nersc.gov/dna/plant/assembly/meraculous2/extras/qqacct>)

2.2 Installation procedure

After you have downloaded and unpacked the source distribution, install the software using the installer script:

```
install.sh <installation directory>
```

..or step by step:

```
mkdir build
```

```
cd build
```

```
cmake -DCMAKE_INSTALL_PREFIX=<installation directory*> ..
```

[Note: The installation directory should be different from the build directory. Omitting the `-DCMAKE_INSTALL_PREFIX` option altogether will install the package into system default locations (`/usr/local/bin`, etc..)]

```
make
```

```
make install
```

Note: If you're planning to run Meraculous in a cluster environment, additional configuration may be required. This documentation assumes that a Grid Engine-type cluster system is in place and can, at the minimum, accept commands `qstat`, `qsub`, and `qacct`, and has the `$SGE_ROOT` variable set to the root directory of the cluster control software. To test if this is the case, type "qstat" on the command line. There should be no errors (the result may be blank, that is ok). Note, however, that even with SGE-like systems, additional changes to the `cluster_submit.sh` wrapper may be needed on the part of the user.

2.3 Test run

The distribution archive contains a small validation dataset to help you confirm that the software has been installed properly. The dataset is meant to be kept its original location, and the validation run should be executed directly from that location. To perform the run, do the following:

```
cd <install_dir>/etc/meraculous/test/pipeline  
bash <install_dir>/bin/run_meraculous.sh -c meraculous.config
```

The run should take no more than 5-10 minutes. When the run completes you will see a new run directory named `run_[date]_[time]`. Inside that directory, open the file `log/meraculous.log`, it should end with the following:

```
meraculous.pl main:: 840> ran to completion successfully
```

If you don't see this then something went wrong. The `meraculous.log` file will have the most detailed record of everything that happened during the run, and it is the first file you should examine when troubleshooting. See section '[Troubleshooting tips](#)' for more details.

3. RUNNING MERACULOUS

3.1 Workflow steps and stages

The Meraculous pipeline is executed in stages which can be run and re-run individually or all at once. Each stage is followed by a cleanup of intermediate files. The level of aggressiveness of this cleanup is controlled by the `-cleanup_level` command line option (for more on this see section '[Executing the run](#)'). Before starting your first assembly, we strongly recommend to familiarize yourself with what happens during each stage, what the key outputs are, and how to monitor the process, which means reading on beyond this chapter, to the end of the document.

`meraculous_import`

- Creates links to the original input sequence files and names these links in a standard format.
- Validates the format and the pairing schema (dual-file or interleaved) of the input sequence files and auto-detects the quality encoding offset (Phred+33 or Phred+64). For more on the types of input data supported, see section '[Input Data Preparation](#)'.
- Divides the input into chunks to allow for parallel processing in the future. This takes two forms. One is at the read level, and for that we simply split the sequence files into chunks of ~500 mb of total sequence. (Note that it is these chunks that will be used from here on and **not** the original files you provided in the config file.) The second type is at the k-mer level. Here we use a sampling of the reads to estimate k-mer frequency rates in the dataset, and based on that define *prefix blocks* to group k-mers in a load-balanced fashion. Because processing the total k-mer space of the dataset is memory-intensive, this load balancing is needed to ensure that the data structures for storing the k-mers require similar amounts of memory. The *number* of blocks is set by the user in the config file (for more on this see section '[Input Data Preparation](#)').

- Generates a human-readable summary of the input dataset extrapolated from a sub-sampling of the reads.

meraculous_mercount

- Counts k-mer frequencies across all libraries that are earmarked for contig generation in the config file, producing a set of `.mercount` files.
- Builds the k-mer frequency histogram which is used to determine the minimum depth cutoff. Users should review this histogram (`mercount.png`) as it can be helpful in identifying features and abnormalities in the dataset, such as contamination or low quality data.

meraculous_mergraph

- For each k-mer that satisfies the minimum depth cutoff Meraculous counts all possible single nucleotide extensions of that k-mer (subject to minimum depth and minimum quality cutoffs) and records them in `.mergraph` files which now provide the basis for the initial contig assembly.

meraculous_ufx

- Classifies each k-mer in the graph as either unique (U), fork (F), or terminal (X), based on the possible single-base extensions of the k-mer in either direction. The main output is a `.UFX` file.

meraculous_contigs

- Loads the UFX information into memory-efficient data structures and builds the initial set of contigs known as UUtigs. A UUtig is built from a tiling path of 'U' mers overlapping by k-1 bp, and terminates on either end when the next extension candidate kmer is either an 'F' type an 'X' type kmer.
- Generates various statistics for the user to review (`UUtigs.fa.stats`). These can be useful to detect problems in contig formation. For example, a lack of long contigs and/or contig sizes failing to add up to the expected genome size may indicate insufficient depth of usable k-mers. If this coincides with a bi-modal k-mer depth distribution (see stage `meraculous_mercount`), then this suggests contamination in the dataset.

[**Note:** In a diploid genome, the total length of all contigs at this stage should be larger than the expected genome size because of the presence of haplotype variant UUtigs]

meraculous_bubble

- Builds depth and size statistics for the UUtigs, which will later be used in selecting contigs for scaffolding.

For diploid assemblies, interrogates ends of UUtigs to detect *bubbles* in the graph caused by diploidy and then orders and transforms qualifying UUtigs into a new type of contigs called "*haplotigs*". Internally, these haplotigs are further categorized as either "*diplotigs*" or "*isotigs*". *Diplotigs* are haplotigs that represent a single haplotype-specific path through a polymorphic region of the genome and are always paired with their alternative haplotype "sister", e.g., `diplotig99_p1 | diplotig99_p2`. All other contigs are termed *isotigs*. These represent homozygous regions as well regions of polymorphism which didn't clearly manifest themselves as bubbles. The latter are expected to be of half the depth compared to the rest of the non-repeat isotigs, and based on this characteristic will be subject to special treatment later on. For more on diploid-aware assembly, see section '[Diploid assembly](#)'.

Important !

Once this stage has completed, we recommend pausing and reviewing the status of the current assembly. Examine the files `mercount.png`, `kha.png`, `UUtigs.fa`, and confirm that the results make sense (see section '[Key output files](#)' on how to interpret these). Based on what you see you may wish to adjust your parameters or even rerun some stages before continuing. For example, an unusually high number of contigs in `UUtigs.fa` and a large low-depth peak in `mercount.png` may mean that your `min_depth_cutoff` parameter should be raised and everything restarted from `meraculous_mergraph`.

For diploid assemblies, you should examine the file `haplotigs.depth.hist.png` and verify that there are two distinct peaks, one at roughly half depth of the other. At this point you may want to check your `bubble_depth_threshold` parameter and adjust it to a value corresponding to the local minimum between the two peaks (if you had originally set it to 0 Meraculous will attempt to auto-detect this threshold).

meraculous_merblast

- Generates depth- and size-filtered contigs to be used from here on. If running with `diploid_mode 1`, all half-depth isotigs are filtered out based on the `bubble_depth_threshold` value (these sequences will be recovered later, during gap closure).
- Maps reads from libraries earmarked for scaffolding (i.e. having the `lib_seq` parameter's `scaffRound` setting set to non-zero) to the contigs, creating `blastMap*.merged` files

meraculous_ono

- Uses mapped read coordinates to "splint" gaps, i.e., link two or more contigs into a scaffold if a single read aligns to the contigs' respective 5' and 3' ends.

[Note: Only libraries earmarked for gap closure by the user are used in splinting. To minimize the effect of chimeras and other library prep artifacts, we strongly recommend using only short insert paired-end Illumina libraries for this purpose.]

- For each library earmarked for scaffolding, determines the actual observed insert size average and std. dev. It is, therefore, crucial that the contigs/scaffolds at this stage are long enough to support enough read pairs mapping to them to result in a reliable insert size distribution.
- Then, using this information, for each set of libraries (as defined by the user with the `lib_seq` parameter's `scaffRound` setting) linkage between contigs is established and scaffolds are built. This process is then iterated for the next `ono` set, bootstrapping from the scaffolds from the previous round.

[**Note:** If running in `diploid_mode 1`, the scaffolding is initially performed using combined linkage info from alternative variant diplotigs, i.e., both variants contribute read pairs to the same link as if they were one and the same contig. Then, using haplotype-specific read mapping info, *phased* variant paths are determined and the scaffold content is corrected in a haplotype consistent manner. As a result, one variant path (typically one with the higher overall depth) will be preserved in a multi-contig scaffold while the individual alternative variants are represented as unlinked, *singleton* scaffolds. A list of these singleton alternative variant scaffolds is also saved. For more on diploid-aware assembly, see section '[Diploid assembly](#)'.]

meraculous_gap_closure

- Based on read pairs mapping to neighboring contigs in a scaffold, a gap size model is generated. Then the scaffold and the gap size information is used to locate the reads suitable for "walking" across the gaps. The actual sequence is drawn from the original fastq files in the form of k-mers.

Meraculous closes the gaps only if enough consistent and high quality sequence exists to bridge it entirely. Currently, no partial extensions of contigs into the gaps are made.

[Note: when running in `diploid_mode 1`, if a gap represents a polymorphic region that had been actively removed earlier (i.e., a half-depth isotig), Meraculous will attempt to walk across it using reads from the more abundant allele].

- The final sequence files (`final.scaffolds.fa`) are built from the scaffolds and the gap closing info, and any scaffolds under 1kb are filtered out. This is considered the final assembly result. When running in `diploid_mode 1`, this file contains only *one* set of variant contigs/scaffolds, while in `diploid_mode 2` both haplomes are represented.

meraculous_final_results

- Generates a brief summary report on the final assembly. For a more in-depth report users should run the standalone script `evaluate_meraculous_run.sh`

3.2 Input data preparation

Meraculous supports the following types in input data:

- Illumina-style sequence in fastq format is the only fully supported input data type at this moment. Using Perl regular expression notation, the supported fastq header formats are:

Illumina versions pre-1.8:

```
/^@\S+:\d+:\d+:\d+:\d+\#[ACTGN0]*)/[12]\s*\S*$/
```

Example: `@071112_SLXA-EAS1_s_4:1:1:672:654/1`

Illumina versions 1.8 and higher:

```
/^@\S+:\d+:\d+:\d+:\d+\s+[12]\:[YN]\:\d+:[ACTGN0]*$/
```

Example: `@HISEQ03:379:C2WP8ACXX:7:1101:1465:2056 2:N:0:ACTTGA`

- For paired libraries, read pairs can be either interleaved within a single file or be split into separate files, e.g. SRA000271.fastq.1 & SRA000271.fastq.2. In the latter case, the reads must be in the same order in both files and in one-to-one correspondence. [**Note:** If your input has been filtered in such a way that some reads have their pairs missing, you will need to edit the files and add dummy reads to take place of the missing pairs]
- All sequence files belonging to a single library should be definable by a single wildcard expression, e.g. SRA0*fastq* (two wildcards are required if reads 1 and 2 are in separate files, e.g. "SRA*fastq1,SRA*fastq2") For more on specifying the inputs see section '[Run configuration](#)'.
- Both Phred+33 and Phred+64 quality encoding schemas are supported. You can have a mix of libraries encoded with either schema, but each individual library must be of one common encoding scheme throughout.
- Both uncompressed and compressed (gzip) fastq files are supported.
- All reads should be at least k-mer size + 1 in length. The upper limit is currently 500 bp
- All reads should be free of adapter or barcode sequence as Meraculous does no explicit trimming or error-correction of the sequence. [**Note:** since low quality data is "naturally" filtered out by Meraculous based on k-mer depth, low complexity sequence, if present in large amounts, will tend to escape this filter]
- Meraculous is optimized for assembly of haploid and diploid genomes only. Running on polyploid or metagenomic datasets and interpreting results can be non-trivial.

3.3 Run configuration

The configuration file contains the parameters guiding the entire assembly process and must be passed to the program with the `-c <file>` argument.

The format of the configuration file is one parameter followed by one or more values. Spaces or tabs can be used as field separators.

[**Note:** an additional optional configuration file named `.meraculous.conf` can be placed in your user home directory and can contain default parameters that are not likely to change for your assembly jobs, e.g. `cluster_queue`, `cluster_slots_per_task`, etc. If there is any redundancy in the parameters in the two files, the run-specific config file takes precedence.]

Core assembly parameters:

The values of these parameters should be set once, at the onset of the run. It's not advisable to change them at subsequent resume/restart attempts.

lib_seq	<p>A multi-argument parameter defining various properties of the input datasets. Normally, for every sequencing library, a separate lib_seq parameter line should be given. <u>The following are the mandatory arguments that must be given as values-only on a single line, in the exact order they're listed here, separated by one or more spaces or tabs. None can be omitted.</u></p> <p>[wildcard] - a bash-style expression (typically a full path) defining the sequence files for a single library. If fwd and rev read pairs are in separate files, then two wildcards should be provided, separated by a comma, without spaces. (See section 'Input data preparation' for more on input data requirements.)</p> <p>[name] - name of the library (caps, numbers)</p> <p>[insertAvg] - estimated average insert size in bp</p> <p>[insertSdev] - estimated std deviation of insert size in bp</p> <p>[avgReadLn] - estimated average read length in bp</p> <p>[hasInnieArtifact] - Whether or not a significant fraction of read pairs is in non-dominant orientation, e.g. "innies" in an "outie" library or vice versa. (0=false, 1=true)</p> <p>[isRevComped] - Whether or not the read pairs are in the "outie" orientation, i.e <-- -->. (0=false, 1=true)</p> <p>[useForContiging]- Whether or not to use this library for initial contig generation. Our recommendation is to use only Paired End (a.k.a. Fragment) libraries for this purpose. (0=false, 1=true)</p> <p>[scaffRound] - Assigns the library to a scaffolding round. Libraries of the same type and similar insert size should be grouped into the same round for the sake of performance. To completely exclude a library from being used in scaffolding, set this to 0.</p>
----------------	--

	<p><i>(positive integers, can be non-consecutive)</i></p> <p>[useForGapClosing] - Whether or not to use this library for gap closing. It's best to use the same libraries that were used for contig generation for this purpose. <i>(0=false, 1=true)</i></p> <p>[5p_wiggleRoom] - During linkage analysis and gap closure, allow reads from this library to have an unaligned 5' end up to this many bp. This option is for cases when a library is known to contain untrimmed adapter sequence. <i>(positive integer, 0 for default behavior [5 bp])</i></p> <p>[downsampleRate] – Use a random subset of all reads (along with pairs) belonging to the library. The value specifies the sampling rate.</p> <p>Example 1: two sets of files - one with fwd reads and the other with reverse reads</p> <pre>lib_seq /path/fastq*.0,/path/fastq*.1 ECO1 200 20 36 0 0 1 1 1 0 0</pre> <p>Example 2: one set of files, all with fwd/rev reads interleaved and up to 20 bp of 5' adapter present</p> <pre>lib_seq /path/to/fastq ECO2 200 20 100 0 0 0 1 0 20 0</pre>
genome_size	<p>Approximate genome size in Gb. Used in estimating depth of read coverage. <i>(positive integer or float)</i></p>
mer_size	<p>The k-mer size to use in meraculous. Must be an odd integer less than the size of the smallest read in the dataset. The optimal k-mer size depends on the quality of the sequence data (error-free read length) and on the genome's repeat content. Picking a k-mer size that's too small means a given sequence will have a lower likelihood of being unique. Too large, and you're increasing the likelihood that the sequence will contain an error, which will cause it to be thrown out entirely by the low-depth filter.</p> <p>There are a number of 3rd party tools available that help you pick the best k-mer size (e.g. kmergenie), or you can calibrate your own rough estimate by running several assemblies with varying k, stopping after the meraculous_mercount stage and examining the mercount.png and kha.png files.</p>
min_depth_cutoff	<p>K-mers less frequent than this cutoff will get excluded from assembly. When assembling you data for the first time, to determine this cutoff, run meraculous.pl through stage 'meraculous_mercount', then look at the k-mer frequency histogram file (mercount.png) and look for a high count (y), low frequency (x) peak that's distinct from the main frequency distribution. This peak represents erroneous k-mers which are best to keep out of the assembly. Set <code>min_depth_cutoff</code> to the low point to the right of the low frequency</p>

	<p>peak and resume the assembly. To auto-detect, set it to 0 <i>(default = 0)</i></p>																														
num_prefix_blocks	<p>Memory usage is optimized by breaking down the DNA search space by prefix, partitioning the k-mers into load-balanced blocks so that each can be processed separately and require similar amount of resources. The greater this number, the less RAM per process. The downside is that greater and greater fraction of run-time will be due to fixed costs like I/O of reading the input, scheduling, etc., so you get less efficiency. Note that only those libraries that will be used in UUtig generation (and therefore turned into k-mer structures) should be considered here.</p> <p>For example, if you have a large genome but want to run on a relatively small server you should set num_prefix_blocks high. This will result in the k-mers being chopped up in more chunks, each small enough to fit into memory when processed separately. If on this server there aren't enough nodes to process all the blocks at once, tasks will simply wait in the queue for their turn.</p> <p>Note: Memory footprint is roughly proportional to the size of the input dataset, i.e., the total sequence used for contiging, but will vary with k-mer size, sequence quality, genome size, and repeat content. The following examples can be used as rough guidelines for estimating the desired number of blocks:</p> <table border="1"> <thead> <tr> <th># of blocks</th> <th colspan="4">Peak memory during k-mer counting (GB)*</th> </tr> <tr> <th></th> <th>H.sapiens (290 Gbp total seq data)</th> <th>S.bicolor - highX (63 Gbp total seq data)</th> <th>S.bicolor - lowX (14 Gbp total seq data)</th> <th>N.fluitans (10 Gbp total seq data)</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>276</td> <td>139</td> <td>23</td> <td>8</td> </tr> <tr> <td>4</td> <td>133</td> <td>42</td> <td>7</td> <td>3</td> </tr> <tr> <td>16</td> <td>24</td> <td>9</td> <td>2</td> <td>0.6</td> </tr> <tr> <td>32</td> <td>12</td> <td>4</td> <td>1</td> <td>0.3</td> </tr> </tbody> </table> <p>* Unlike k-mer counting, the UUtig assembly stage cannot be partitioned this way and thus has fixed memory requirements which can be the limiting factor for large genomes. For more, see section 'Hardware considerations'</p>	# of blocks	Peak memory during k-mer counting (GB)*					H.sapiens (290 Gbp total seq data)	S.bicolor - highX (63 Gbp total seq data)	S.bicolor - lowX (14 Gbp total seq data)	N.fluitans (10 Gbp total seq data)	1	276	139	23	8	4	133	42	7	3	16	24	9	2	0.6	32	12	4	1	0.3
# of blocks	Peak memory during k-mer counting (GB)*																														
	H.sapiens (290 Gbp total seq data)	S.bicolor - highX (63 Gbp total seq data)	S.bicolor - lowX (14 Gbp total seq data)	N.fluitans (10 Gbp total seq data)																											
1	276	139	23	8																											
4	133	42	7	3																											
16	24	9	2	0.6																											
32	12	4	1	0.3																											

Optional assembly parameters

diploid_mode	<p>Specifies ways to handle diploidy. 0 - haploid assembly; variant/bubble detection is turned off. 1 – diploid with low polymorphism rate (< 1 per <i>k bp</i>); The general aim is to suppress variability and to consistently reproduce a single haplotype. Simplifies assembly graph by “squashing” variant-induced</p>
--------------	--

	<p>“bubbles” during scaffolding while keeping all variants at the contig level.</p> <p>2 – diploid with high polymorphism rate; Relies on significant sequence divergence to reproduce <i>both</i> haplotypes. Simplifies assembly graph by maintaining two alternative “haplo-paths” .</p> <p>Related parameters: <code>bubble_depth_threshold</code> <code>no_strict_haplotypes</code></p> <p>See ‘ Technical Notes/Diploid assembly’ for more info.</p> <p>Triploid or higher ploidy genomes are not supported at this point. (<i>default = 0</i>)</p>
<p><code>bubble_depth_threshold</code></p>	<p><i>Valid only with diploid_mode 1 & 2.</i></p> <p>After bubble resolution, some fraction of the bubble-free contigs (termed isotigs) still contains haplotype variants that couldn't be recognized/resolved. In the contig depth distribution at that stage (see file <code>haplotigs.depth.hist</code>) these contigs form a second peak at roughly 1/2 the depth of the main, non-polymorphic depth peak. This cutoff sets the depth threshold that will help distinguish these variant isotigs from non-variant ones. It should be set to the through between the two peaks.</p> <p>To auto-detect, set it to 0 (<i>default = 0</i>)</p>
<p><code>no_strict_haplotypes</code></p>	<p><i>Valid only with diploid_mode 1 & 2</i></p> <p>Lifts the requirement that all haplotype variants within a single diplotig are phased via common clone linkage. Turning this parameter on may help increase the final scaffold/contig N50, but at the expense of more frequent haplotype crossover. Running in <code>diploid_mode 1</code> with this parameter enabled would effectively mimic the behavior of older versions of Meraculous (v.2.0.5 in particular). (<i>0 1; default=0</i>)</p>
<p><code>mergraph_depth_pct_cutoff</code></p>	<p><i>For meta-genomic assemblies only.</i></p> <p>K-mer extension candidates' counts (i.e. depth) are evaluated as percentages of all candidates' counts combined. This has the effect of normalizing the minimum depth requirement to the organism's abundance in the sample. If used, it's best to keep this value close to your data set's expected sequencing error rate multiplied by the k-mer size (i.e. the probability of having an erroneous kmer). With higher values you increase the risk of throwing out legitimate variants and/or repeat boundaries. Note that <code>min_depth_cutoff</code> is still valid and serves as the hard "floor". (<i>float; range: 0.0 – 1.0</i>)</p>

no_read_validation	Set to 1 to skip validation of input fastq reads' headers, sequence, and q-scores. This will speed up the processing of reads in stage meraculous_import. We recommend using this option only when re-running with a previously validated dataset. <i>(0 1; default=0)</i>
fallback_on_est_insert_size	If the program can't determine the actual <u>assembly-based</u> insert size average for a library, this option will allow it to continue using the initial estimates provided by user (see lib_seq). Use this parameter as the last resort only! <i>(0 1; default=0)</i>
gap_close_aggressive	Close gaps more aggressively, accepting closures that might violate the estimated gap size. <i>(0 1; default=0)</i>
gap_close_rpt_depth_ratio	If the average k-mer depth of a given scaffold exceeds the overall modal peak depth for all scaffolds by more than this factor the scaffold is assumed to be a collapsed repeat and is excluded from consideration during meraculous_gap_closure. Raise this cutoff if you know the depth distribution in your dataset to be highly irregular, e.g. it's a metagenome or the like. <i>(default = 2.0)</i>

Resource utilization parameters

local_num_procs	<i>Valid only when 'use_cluster' is off.</i> Number of processors to occupy simultaneously when running jobs locally. This should normally equal the number of available cpus. For non-threaded processes this sets the maximum number of processes executed in parallel. For threaded processes this sets the number of threads. <i>(default = 1)</i>
num_procs_<stage>	Can be used to override 'local_num_procs' on a per-stage basis
local_max_memory	Set a memory limit (GB) for local processes. Where possible, data will be partitioned to fit under this limit.
local_max_retries	Number of retries before failure for local jobs
use_cluster	Specifies whether to use a cluster for job submissions. Requires an SGE-like cluster system to be configured and ready to accept

	scheduling and monitoring commands like qsub, qacct, and qstat.
cluster_num_nodes	Number of available cluster nodes (can be approximate). This is used for partitioning the data into appropriately sized chunks to be processed in parallel. Combined with parameters 'cluster_ram_request' and 'cluster_slots_per_task', it can be used to control the granularity of job sets. <i>(default =1 The default setting means data will not be partitioned if it can fit under the memory limit set by cluster_ram_request !!!)</i>
cluster_slots_per_task	Number of slots (cores) to allocate for multi-threaded tasks when submitting to the cluster. Threaded tasks will spin off this many threads each. For non-threaded tasks this option is ignored. Typically, you would set this to the number of cores on the smallest available node. If the nodes are shared and are heavily used, you may want to refrain from occupying all the CPUs on them.
cluster_ram_request	Amount of memory (GB) to request <i>per task</i> on the cluster. If you request too much your process will have few available nodes to use, too little and your process will get killed by the cluster scheduler if it exceeds this limit.
cluster_ram_<stage>	Can be used to override the above cluster_ram_request on a per-stage basis.
cluster_walltime	Walltime limit for cluster tasks. Must be specified as hh:mm:ss
cluster_walltime_<stage>	Can be used to override the above cluster_walltime limit on a per-stage basis
cluster_max_retries	Number of retries before failure for cluster jobs
cluster_project	Name of project to which cluster jobs will be assigned. (This is needed only when your cluster uses project-based allocation schema)
cluster_queue	Name of queue to which cluster jobs will be assigned.

3.4 Executing the run

An assembly run that includes all stages, with intermediate data cleaned up at the end, is executed as follows:

```
run_meraculous.sh -c <config file>
```

This will generate a run directory named `run_<date>_<time>`. Inside the run directory, files `log/info.log`, `log/error.log`, and `log/meraculous.log` contain the detailed record of the assembly process. One way to monitor a running assembly is to watch the `info.log` or `meraculous.log` files "live" with `'tail -f'`. See section '[Logs](#)' for more on these files.

Other command line options:

<code>-dir <></code> <code>-label <></code>	For new assembly runs, <code>-dir</code> lets you name your output run folder, while <code>-label</code> will attach the specified string as a prefix to the default name. For resuming or restarting an existing run, you're required to provide the run folder with the <code>-dir</code> option.
<code>-archive</code>	When restarting a stage from the beginning, save any existing stage directories under the subdirectory <code>old/</code>
<code>-debug</code>	Record all commands and additional troubleshooting information in the file <code>logs/meraculous.log</code> (Currently on by default)
<code>-cleanup_level <0 1 2></code>	Determines how aggressively the pipeline should clean up intermediate data after each stage. The possible arguments are: 0 - Do not delete any intermediate outputs. This will provide the ability to go back and examine full stage outputs or restart from any stage at any point in the future. The flip side is that with this option, the disk space footprint may be several times the size of the input dataset. 1 (default) - Delete files that are not used in any of the subsequent stages and that are generally not informative to the user. You will still be able to rerun any stage individually. 2 - Delete as much as possible, as soon as possible. With this option you will not be able to rerun the stages individually once they have completed. The use of this option should be reserved to cases where

	disk space is tight and when you're confident that parts of the assembly will not need to be rerun.
<code>-restart</code>	Restart a previously failed run from the last successful stage
<code>-resume</code>	Similar to <code>-restart</code> , but but preserves any partial results from successfully completed processes within the stage
<code>-step</code>	Execute one stage and stop
<code>-start <stage></code>	Re-run starting with this stage (requires <code>-resume</code> or <code>-restart</code>)
<code>-stop <stage></code>	Stop after this stage
<p>Invalid combinations:</p> <p><code>-restart</code> with <code>-resume</code> <code>-start</code> without <code>-restart/resume</code> <code>-archive</code> without <code>-restart</code></p> <p>See more about <code>-restart</code> and <code>-resume</code> options in section ‘Working in stages’.</p>	

3.5 Working in stages

A Meraculous run consists of stages that can be executed and re-executed separately or as a set. Each stage reloads the parameters from the main user-specified config file. It also loads parameters from local parameter files created by preceding stages and writes local parameters for subsequent stages to use.

Note: Users are free to change the parameters in the config file between restart/resume attempts, however, the following parameters are considered "set in stone" and cannot be changed, removed, or added after the onset on the run:

```
mer_size
num_prefix_blocks
local_num_procs
cluster_slots_per_task
```

Upon successful completion a checkpoint file is created inside the checkpoints/ directory, which signals to Meraculous.pl that the stage has been completed, and the next time the pipeline is executed that stage will be skipped.

If for some reason the run exits in the middle of a stage, the user has an option to resume from where the previous attempt left off (presumably after making necessary corrections). Various milestones in the stage's progress are marked by "resume checkpoints" (not to be confused with checkpoint files described above). Once the pipeline determines the stage it needs to execute, it will jump to the specified resume checkpoint inside that stage, if one exists. Use the `-resume` option to enable this behavior.

If, instead of resuming, you wish to rerun an entire stage from the beginning or rerun starting with a certain stage, use the `-restart` option combined with `-start [stage]` and `-stop [stage]` to specify which stage to start and end the run with. The `-restart` option will cause the deletion of all previously completed stages from the '-start' stage and onward. If you wish to save those, use the `-archive` option, and they will be moved to a newly created `old/` subdirectory in your run.

If you wish to step through the stages, stopping after each one, use `-step` option

Note: The ability to resume the assembly process from a middle point is the main reason why Meraculous writes a large amount of intermediate data to disk. Setting `-cleanup_level` to 0 will cause many of these files to be deleted as soon as they're used and thus prohibits the 'restart' behavior while 'resume' should still be possible.

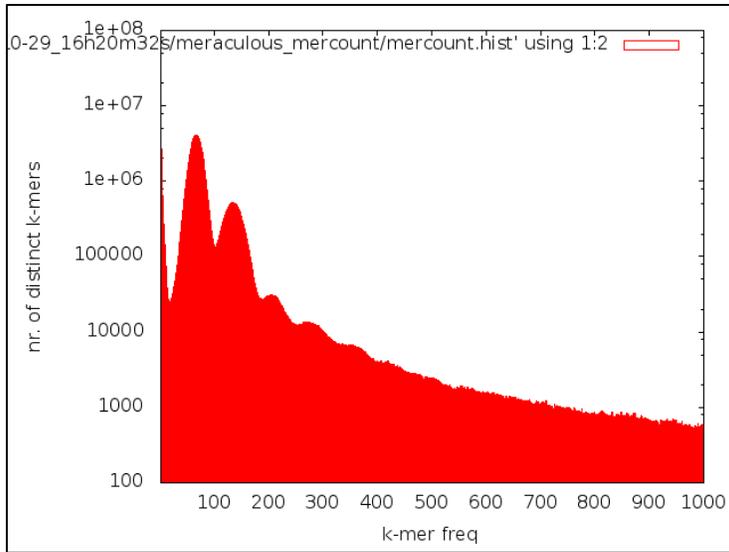
4. EVALUATING RESULTS & TROUBLESHOOTING

4.1 Key output files

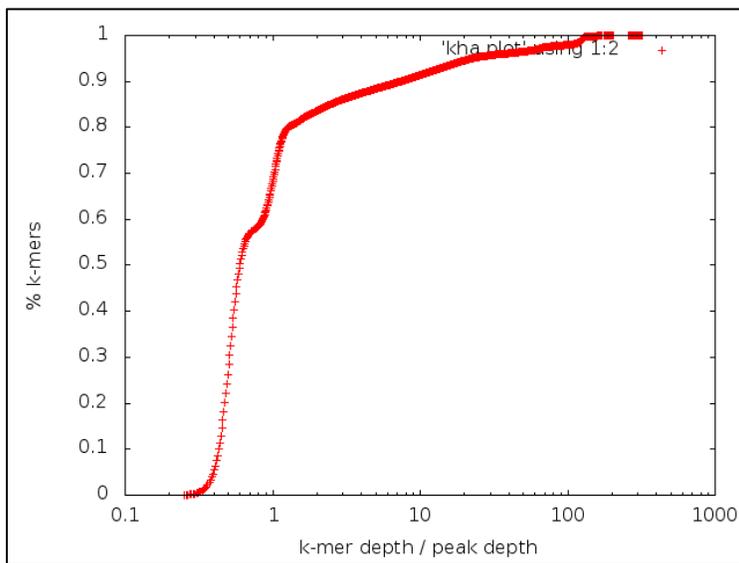
- `meraculous_final_results/SUMMARY.txt`: Brief summary of assembly inputs and results (for creating a more detailed report see [Run evaluation script](#))
- `meraculous_final_results/final.scaffolds.fa`: The final set of scaffolds over 1 kb in total length in Fasta format. Gaps between contigs are filled with stretches of Ns whose number corresponds to the estimated gap size.
Note: If you ran with `diploid_mode` set to 2, this set of scaffolds will include any alternative variant contigs as well.
- `meraculous_merblast/contigs.fa`: The initial set of UUtigs (or haplotigs if diploid) that were used as input to the scaffolding stages. The total size of the contigs in this file should be fairly close to the estimated genome size, with the remainder assumed to be in gaps that are to be spanned and filled in later stages. The file is in Fasta format; run the included `fasta_stats` program on this file to get a breakdown on contig size distribution.

Main genome contig total: 3190345					
Main genome contig sequence total: 2691.5 MB (-> 0.0% gap)					
Main genome contig N/L50: 284358/2.5 KB					
Minimum Scaffold Length	Number of Scaffolds	Number of Contigs	Total Scaffold Length	Total Contig Length	Scaffold Contig Coverage
All	3,190,345	3,190,345	2,691,458,526	2,691,458,526	100.00%
1 kb	713,602	713,602	2,037,767,767	2,037,767,767	100.00%
2.5 kb	290,691	290,691	1,361,689,624	1,361,689,624	100.00%
5 kb	89,217	89,217	661,837,734	661,837,734	100.00%
10 kb	11,367	11,367	143,937,099	143,937,099	100.00%
25 kb	47	47	1,298,589	1,298,589	100.00%
50 kb	0	0	0	0	0.00%

- `meraculous_mercount/mercount.png`: Histogram of k-mer abundance across the entire dataset used for contig generation. This can be useful for identifying various anomalies and trends with the dataset. Normally you should be able to identify the single peak corresponding to unique genomic k-mers (two peaks if diploid), a low-depth peak for the erroneous k-mer population, and additional high-depth peaks for k-mers representing genomic repeats. The histogram plot can also be useful when evaluating the chosen k-mer size. The optimal k is usually the largest k that still allows to identify and cleanly separate the low-depth peak.

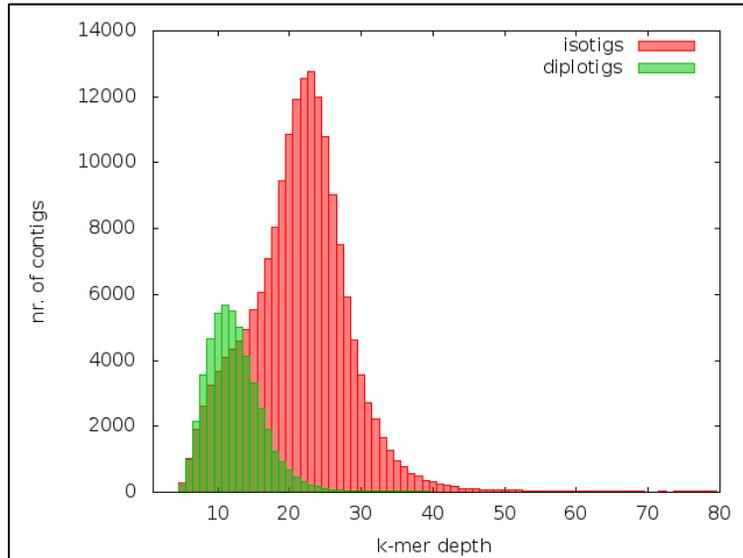


- `meraculous_mercount/kha.png`: This plot shows the cumulative fraction of all k-mers in the dataset as a function of k-mer depth, which can be useful in identifying distinct k-mer “populations”, e.g. repeats, polymorphic regions, contamination.



- `meraculous_bubble/haplotigs.depth.hist.png`: (For diploid assemblies only) This plot shows the distribution of contigs’ weighted kmer depth after bubble detection, with isotigs and diplotigs plotted separately. It can help greatly when troubleshooting scaffolding issues since it reveals how well polymorphisms are being identified and treated. It can also help verify the selected `bubble_depth_threshold` value. For instance, a very low *diplotigs* peak completely overlapping a high *isotigs* peak would signal poor detectability of bubbles as a likely consequence of a very high polymorphism rate. On the other hand, a low *diplotigs* peak

alongside of a high *isotigs* peak at twice the depth suggests low polymorphism rate. As long as you can identify the *diplotigs* peak at $\frac{1}{2}$ the depth of the main *isotigs* peak, set `bubble_depth_threshold` at the midpoint between them (around 17 in the example below).



- `meraculous_ono/*.srf`: Scaffold reference files from each scaffolding round. The last digit in the file name specifies the scaffolding round; the first digit specifies the minimum number of links cutoff used to form the scaffolds. These files allow mapping of the original contigs to the current scaffolds.

4.2 Logs

The log files are critical for both monitoring the run progress and troubleshooting the results. You will find yourself looking at these files a lot, so it pays to get familiar with them early.

`info.log` - This is a concise record of the run, designed to inform a non-expert user of the key events during the run and raise a basic "Errors encountered" flag in case of a run failure. During a run, a good practice is watch this file in real time (e.g. `'tail -f run_abc/log/info.log'`)

`meraculous.log` - This is the most complete and verbose record of the process, designed for troubleshooting a failed run and is generally geared to a more expert user. In addition to the general messages from `info.log`, it records all commands the pipeline executes as local system calls,

parallel job sets, resource usage and timing statistics. It will also capture all errors thrown by the Meraculous pipeline with information on which component threw the error.

*.err files - Errors from standalone programs executed by the pipeline will normally be captured in that program's standard error file. These files are always named and placed according to the program's intended output file and are specified as part of the command record in the meraculous.log file. For example, if Meraculous starts a oNo4.pl process which is meant to output a file p3.2.srf, then the standard error will be captured in the file p3.2.srf.err in the same directory. See more on this in section '[Troubleshooting tips](#)'.

4.3 Run evaluation script

The bash script `evaluate_meraculous_run.sh` collects various stats from the current run including any restart/resume attempts, and could be very useful in troubleshooting and understanding what happened at the different assembly stages (see below). We encourage users to run this script after the entire run is finished. It does not get executed automatically.

4.4 Troubleshooting tips

Errors and abnormal termination

When a run aborts prematurely, the error that caused the exit appears at the end of the meraculous.log and error.log files, e.g.

```
2014/03/18 13:31:12 meraculous.pl main::run_meraculous_gap_closure 2705> No fastq files
found for library ECO
2014/03/18 13:31:12 meraculous.pl main:: 711> Stage meraculous_gap_closure failed (0.001786
seconds)
2014/03/18 13:31:12 meraculous.pl main:: 771> ERRORS ENCOUNTERED!
2014/03/18 13:31:12 meraculous.pl main:: 775> Total run time: 0.832589 seconds.
```

Often, in order not to overload the logs with repetitive error messages, the core executables write errors and other troubleshooting info to stderr which gets captured by the pipeline into

corresponding .err files inside the stage directories. If meraculous.log doesn't provide enough information about the root cause of the early exit, look for the last command that was attempted and see where the stderr was redirected to. Then review the messages in that file. For example, meraculous.log may have the following entry:

```
2014/03/26 12:08:49 M_Job_Set.pm M_Job_Set::run_job_set_local 765> Local command returned a
non-zero exit status! Check stderr outputs for more clues!
Return value (65280) Command ( perl gapPlacer.pl -b
../meraculous_merblast/blastMap.ECO.f0.merged -m 19 -i 215:10 -s
../meraculous_ono/ROUND_1/p7.1.srf
-f ../meraculous_merblast/ECO.fastq.info.0 -c ../meraculous_merblast/contigs.fa -F 10 >
gapData.ECO.0 2> gapData.ECO.0.err)
```

The file gapData.ECO.0.err will contain more info on what went wrong:

```
$ tail gapData.ECO.0.err
.
.
.
Total reads placed in gaps = 487780 (aligned) + 1090614 (projected)
Reading sequence file ../meraculous_import/ECO.fastq.0_00001...
Couldn't open ../meraculous_import/ECO.fastq.0_00001
```

When running in *cluster* mode, the actual submission commands and any submission-related errors returned are captured in the files linkedScript.template.submit.<date>.err inside the stage directories. These can be useful in troubleshooting cluster compatibility issues. Submission commands are formed by the cluster_submit.sh wrapper, and that is typically the script to edit if one needs to modify the syntax of the submission (this is different from the job monitoring commands which are issued and logged directly by the pipeline)

```
$ cat linkedScript.template.submit.20140630-122647.err

qsub -v MERACULOUS_ROOT -cwd -r n -b n -S /bin/bash -w e -j y -N gapClosure -o run_2014-
06-27_15h58m30s/meraculous_gap_closure/JOB_SET_LOG.gapClosure
-P plant-assembly.p -l h_rt=00:30:00 -l ram.c=1G -l h_vmem=1G -t 1-1 run_2014-06-
27_15h58m30s/meraculous_gap_closure/linkedScript.template
```

Problems with assembly results

The primary causes of poor assemblies are usually sequencing library quality and/or settings, depth of coverage, and sequencing artifacts (e.g. untrimmed adapter). The output of

`evaluate_meraculous_run.sh` can give clues about the input data. The section **MERCOUNTS** of the report, for instance, reports the weighted average depth based on the k-mer count. If the estimate is below 15x, the assembly quality is likely to suffer.

```
Total 31-mers (over 3x) : 219691822
Total unique sequeces (over 3x) : 9308288
Weighted average 31-mer depth : 23.6
```

You should also refer to `mercount.png` and `kha.png` files for the actual distribution plots.

If the coverage is sufficient, check how well each library did during scaffolding. The section **LIBRARY MAPPING ANALYSIS** can point to a library whose data is getting rejected for a specific reason.

```
library      total_mapped      hits_omitted(rate)      total_spans
ECO1         6383564           6035869 (.915)         74585
ECO2         20522013          734836 (.035)         317968

Read mappings omitted for following reasons:
library      truncated_align      singleton      minlen
ECO1         4036475              499394        0
ECO2         510942               223894        0
```

5. TECHNICAL NOTES

5.1 Diploid assembly

Meraculous offers two methods for handling allelic variation in diploid assembly. Both methods begin by identifying distinct diploid variant signatures in the contig graph, termed *bubbles*. Here we define a bubble as pairs of UUtigs that share a common unique k-mer extension at both ends. In other words, at both ends, bubble-UUtigs terminate immediately prior to the start of a homozygous UUtig. Bubbles are then linked into chains of UUtig-(bubble-UUtig)ⁿ which are fused to produce longer contiguous sequences. We refer to these new sequences as “diplotigs,” which represent uncontested, heterozygosity-containing stretches of the genome. For every bubble chain, exactly two *phased* diplotigs are produced, representing the two allelic haplotypes.

The phasing of the haplotypes is accomplished by relying on common reads or read pairs mapping to neighboring bubble UUtigs in a multi-bubble chain. The chain is traversed one bubble at a time, and the corresponding diplotigs are extended if, and only if, the next bubble can be reliably phased with

the current one. Otherwise, the extension process is terminated and a pair of truncated diplotigs is reported. A new pair is then initiated at the point of termination, and the traversal of the bubble chain can continue (this restriction is lifted if the parameter `no_strict_haplotypes` is set to 1). The result is two sets of diplotigs (named in a pairwise fashion to help identify the alternative variants, e.g., `diplotig99_p1` & `diplotig99_p2`), representing all the variants in the genome that could be detected as bubbles in the contig graph.

The differences between the two diploid modes lie primarily in how the diplotigs are represented and handled in the scaffolding and gap-closing stages that follow, and the choice between them ultimately depends on the frequency and the nature of allelic variation in the given genome. By our definition, UUtigs must terminate when the next k-mer extension is either an X- or and F-type kmer, meaning a physical coverage gap or a fork. When the polymorphic rate is low with respect to the chosen kmer size ($\text{rate} < 1/k$), we can expect most variant-containing UUtigs to be short and terminate at both ends at an F-type kmer which represents the end of a variant region, meaning most variant regions will manifest themselves as bubbles. However, when the polymorphic rate increasing above $1/k$, variant-containing UUtigs grow progressively larger (due to compounding effects of multiple alleles) and have a much higher chance of terminating for reasons other than a variant-induced F-type kmer, i.e., the cause of termination will more likely be a physical gap, a repeat- or error-induced fork. This ultimately leads to more and more “uncaptured” variants which would greatly hinder further scaffolding and gap closure. Simply filtering these ‘isotigs’ out by depth is no longer a viable option since a big portion of the genome can end up in this type of contigs. To account for this scenario, which is becoming more and more common with increasing read lengths and quality (and therefore, higher optimal k), the second diploid assembly mode was developed which actually takes advantage of highly divergent haplotypes. The two modes behave as follows:

Diploid mode 1: Only one contig of each variant diplotig pair is used in scaffolding while the alternative variant is left as a singleton. This simplifies the assembly graph and leads to greater long-range contiguity, and is the method to use for most common diploid scenarios. Read mapping information which is used for scaffolding is generated for *both* variant diplotigs but is used in an aggregate fashion when establishing contig linkage, meaning, the bubble is linked as a whole, even though only one of the variants gets incorporated in a longer scaffold.

After the scaffolds are built, the unincorporated singleton diplotig-sisters can be swapped in to replace the initially chosen variants as part of the haplotype phasing process which aims to produce haplotype-consistent scaffolds. Gap closing then proceeds normally, using all mapped reads that are thought to extend or project into gaps. At the end, in the file `final.scaffolds.fa` the singleton alternative variants are removed, i.e., each scaffold is meant to represent a single, phased, haplotype. The file `final.scaffolds.fa.unfiltered` contains *all* variant scaffolds, regardless of size.

Diploid mode 2: This method can be thought of as the polar opposite of diploid mode 1. When the polymorphism rate is excessively high, many variants no longer fit the “bubble” model. Instead, it makes more sense to treat the assembly as two haploid genomes, i.e. preserve and take advantage of the haplotype differences to gradually build up haplotype-specific scaffolds while identifying and treating *non*-variant regions in a way that removes linkage ambiguity. Although we still try to detect

and resolve as many bubbles as possible during the `meraculous_bubble` stage, here it's the non-variant regions that are simpler to detect heuristically since, in a highly polymorphic genome, they will tend to be smaller in size and display a well-recognizable kmer depth profile. We, therefore, don't attempt to actively phase individual variant contigs, leaving that to happen naturally since most contigs are expected to connect only to contigs of the same haplotype. During gap closing, we avoid anchoring our walks on reads that mapped to non-variant regions, thus allowing haplotype consistency to be preserved while bridging gaps. At the end, the file `final.scaffolds.fa` contains **both** haplotypes and, in cases of highly polymorphic genomes, is expected to be roughly twice the size of the diploid genome.

When assembling a large diploid genome where variant frequency may be far from uniform, it makes sense to try both modes. In either case, setting the `bubble_depth_threshold` parameter (see section '[Run configuration](#)') appropriately will be a key to improving the assembly. Use the pipeline's options `-restart` and `-start meraculous_bubble` to retry the assembly in a new diploid mode (the mode is set in the configuration file).

6. CITING AND FEEDBACK

If you use Meraculous in your research, please cite: []

We would also like to reference back to your publications on our site. Please email the reference, the name of your lab, department and institution to egoltsman@lbl.gov.

We welcome your comments and bug reports. Please send them along with the `meraculous.log` file to egoltsman@lbl.gov.