canu Documentation

Release 1.6

Adam Phillippy, Sergey Koren, Brian Walenz

Aug 14, 2017

Contents

CHAPTER 1

Canu Quick Start

Canu specializes in assembling PacBio or Oxford Nanopore sequences. Canu operates in three phases: correction, trimming and assembly. The correction phase will improve the accuracy of bases in reads. The trimming phase will trim reads to the portion that appears to be high-quality sequence, removing suspicious regions such as remaining SMRTbell adapter. The assembly phase will order the reads into contigs, generate consensus sequences and create graphs of alternate paths.

For eukaryotic genomes, coverage more than 20x is enough to outperform current hybrid methods, however, between 30x and 60x coverage is the recommended minimum. More coverage will let Canu use longer reads for assembly, which will result in better assemblies.

Input sequences can be FASTA or FASTQ format, uncompressed or compressed with gzip (.gz), bzip2 (.bz2) or xz (.xz). Note that zip files (.zip) are not supported.

Canu can resume incomplete assemblies, allowing for recovery from system outages or other abnormal terminations.

Canu will auto-detect computational resources and scale itself to fit, using all of the resources available and are reasonable for the size of your assembly. Memory and processors can be explicitly limited with with parameters maxMemory and maxThreads. See section *Execution Configuration* for more details.

Canu will automaticall take full advantage of any LSF/PBS/PBSPro/Torque/Slrum/SGE grid available, even submitting itself for execution. Canu makes heavy use of array jobs and requires job submission from compute nodes, which are sometimes not available or allowed. Canu option `useGrid=false` will restrict Canu to using only the current machine, while option `useGrid=remote` will configure Canu for grid execution but not submit jobs to the grid. See section *Execution Configuration* for more details.

The *Canu Tutorial* has more background, and the *Canu FAQ* has a wealth of practical advice.

Assembling PacBio or Nanopore data

Pacific Biosciences released P6-C4 chemistry reads for Escherichia coli K12. You can download them from their original release, but note that you must have the SMRTpipe software installed to extract the reads as FASTQ. Instead, use a FASTQ format 25X subset (223MB). Download from the command line with:

```
curl -L -o pacbio.fastq http://gembox.cbcb.umd.edu/mhap/raw/ecoli_p6_25x.filtered.
↪fastq
```

There doesn't appear to be any "official" Oxford Nanopore sample data, but the Loman Lab released a set of runs, also for Escherichia coli K12. This is early data, from September 2015. Any of the four runs will work; we picked MAP-006-1 (243 MB). Download from the command line with:

```
curl -L -o oxford.fasta http://nanopore.s3.climb.ac.uk/MAP006-PCR-1_2D_pass.fasta
```

By default, Canu will correct the reads, then trim the reads, then assemble the reads to unitigs. Canu needs to know the approximate genome size (so it can determine coverage in the input reads) and the technology used to generate the reads.

For PacBio:

```
canu \
 -p ecoli -d ecoli-pacbio \
 genomeSize=4.8m \
 -pacbio-raw pacbio.fastq
```

For Nanopore:

```
canu \
 -p ecoli -d ecoli-oxford \
 genomeSize=4.8m \
 -nanopore-raw oxford.fasta
```

Output and intermediate files will be in directories 'ecoli-pacbio' and 'ecoli-nanopore', respectively. Intermeditate files are written in directories 'correction', 'trimming' and 'unitigging' for the respective stages. Output files are named using the '-p' prefix, such as 'ecoli.contigs.fasta', 'ecoli.contigs.gfa', etc. See section *Outputs* for more details on outputs (intermediate files aren't documented).

Assembling With Multiple Technologies and Multiple Files

Canu can use reads from any number of input files, which can be a mix of formats and technologies. We'll assemble a mix of 10X PacBio reads in two FASTQ files and 10X of Nanopore reads in one FASTA file:

```
curl -L -o mix.tar.gz http://gembox.cbcb.umd.edu/mhap/raw/ecoliP6Oxford.tar.gz
tar xvzf mix.tar.gz

canu \
 -p ecoli -d ecoli-mix \
 genomeSize=4.8m \
 -pacbio-raw pacbio.part?.fastq.gz \
 -nanopore-raw oxford.fasta.gz
```

Correct, Trim and Assemble, Manually

Sometimes, however, it makes sense to do the three top-level tasks by hand. This would allow trying multiple unitig construction parameters on the same set of corrected and trimmed reads, or skipping trimming and assembly if you only want correced reads.

We'll use the PacBio reads from above. First, correct the raw reads:

```
canu -correct \
  -p ecoli -d ecoli \
  genomeSize=4.8m \
  -pacbio-raw  pacbio.fastq
```

Then, trim the output of the correction:

```
canu -trim \
  -p ecoli -d ecoli \
  genomeSize=4.8m \
  -pacbio-corrected ecoli/ecoli.correctedReads.fasta.gz
```

And finally, assemble the output of trimming, twice, with different stringency on which overlaps to use (see *correctedErrorRate*):

```
canu -assemble \
  -p ecoli -d ecoli-erate-0.039 \
  genomeSize=4.8m \
  correctedErrorRate=0.039 \
  -pacbio-corrected ecoli/ecoli.trimmedReads.fasta.gz

canu -assemble \
  -p ecoli -d ecoli-erate-0.075 \
  genomeSize=4.8m \
  correctedErrorRate=0.075 \
  -pacbio-corrected ecoli/ecoli.trimmedReads.fasta.gz
```

Note that the assembly stages use different '-d' directories. It is not possible to run multiple copies of canu with the same work directory.

Assembling Low Coverage Datasets

We claimed Canu works down to 20X coverage, and we will now assemble a 20X subset of S. cerevisae (215 MB). When assembling, we adjust *correctedErrorRate* to accomodate the slightly lower quality corrected reads:

```
curl -L -o yeast.20x.fastq.gz http://gembox.cbcb.umd.edu/mhap/raw/yeast_filtered.20x.
↪fastq.gz

canu \
 -p asm -d yeast \
 genomeSize=12.1m \
 correctedErrorRate=0.075 \
 -pacbio-raw yeast.20x.fastq.gz
```

Consensus Accuracy

Canu consensus sequences are typically well above 99% identity. Accuracy can be improved by polishing the contigs with tools developed specifically for that task. We recommend Quiver for PacBio and Nanopolish for Oxford Nanpore data. When Illumina reads are available, Pilon can be used to polish either PacBio or Oxford Nanopore assemblies.

CHAPTER 2

Canu FAQ

What resources does Canu require for a bacterial genome assembly? A mammalian assembly?

Canu will detect available resources and configure itself to run efficiently using those resources. It will request resources, for example, the number of compute threads to use, Based on the `genomeSize` being assembled. It will fail to even start if it feels there are insufficient resources available.

A typical bacterial genome can be assembled with 8GB memory in a few CPU hours - around an hour on 8 cores. It is possible, but not allowed by default, to run with only 4GB memory.

A well-behaved large genome, such as human or other mammals, can be assembled in 10,000 to 25,000 CPU hours, depending on coverage. A grid environment is strongly recommended, with at least 16GB available on each compute node, and one node with at least 64GB memory. You should plan on having 3TB free disk space, much more for highly repetitive genomes.

Our compute nodes have 48 compute threads and 128GB memory, with a few larger nodes with up to 1TB memory. We develop and test (mostly bacteria, yeast and drosophila) on laptops and desktops with 4 to 12 compute threads and 16GB to 64GB memory.

How do I run Canu on my SLURM / SGE / PBS / LSF / Torque system?

Canu will detect and configure itself to use on most grids. You can supply your own grid options, such as a partition on SLURM or an account code on SGE, with `gridOptions="<your options list>"` which will passed to every job submitted by Canu. Similar options exist for every stage of Canu, which could be used to, for example, restrict overlapping to a specific partition or queue.

To disable grid support and run only on the local machine, specify `useGrid=false`

My run stopped with the error `'Failed to submit batch jobs'`

The grid you run on must allow compute nodes to submit jobs. This means that if you are on a compute host, `qsub/bsub/sbatch/etc` must be available and working. You can test this by starting an interactive compute session and running the submit command manually (e.g. `qsub` on SGE, `bsub` on LSF, `sbatch` on SLURM).

If this is not the case, Canu **WILL NOT** work on your grid. You must then set `useGrid=false` and run on a single machine. Alternatively, you can run Canu with `useGrid=remote` which will stop at every submit command, list what should be submitted. You then submit these jobs manually, wait for them to complete, and run the Canu command again. This is a manual process but currently the only workaround for grids without submit support on the compute nodes.

What parameters should I use for my reads?

Canu is designed to be universal on a large range of PacBio (C2, P4-C2, P5-C3, P6-C4) and Oxford Nanopore (R6 through R9) data. Assembly quality and/or efficiency can be enhanced for specific datatypes:

Nanopore R7 1D and Low Identity Reads With R7 1D sequencing data, and generally for any raw reads lower than 80% identity, five to ten rounds of error correction are helpful. To run just the correction phase, use options `-correct corOutCoverage=500 corMinCoverage=0 corMhapSensitivity=high`. Use the output of the previous run (in `asm.correctedReads.fasta.gz`) as input to the next round.

Once corrected, assemble with `-nanopore-corrected <your data> correctedErrorRate=0.3 utgGraphDeviation=50`

Nanopore R7 2D and Nanopore R9 1D Increase the maximum allowed difference in overlaps from the default of 4.5% to 7.5% with `correctedErrorRate=0.075`

Nanopore R9 2D and PacBio P6 Slightly decrease the maximum allowed difference in overlaps from the default of 4.5% to 4.0% with `correctedErrorRate=0.040`

Early PacBio Sequel Based on exactly one publically released *A. thaliana* dataset, slightly decrease the maximum allowed difference from the default of 4.5% to 4.0% with `correctedErrorRate=0.040 corMhapSensitivity=normal`. For recent Sequel data, the defaults are appropriate.

Nanopore R9 large genomes Due to some systematic errors, the identity estimate used by Canu for correction can be an over-estimate of true error, inflating runtime. For recent large genomes (>1gbp) we've used `'corMhapOptions=--threshold 0.8 --num-hashes 512 --ordered-sketch-size 1000 --ordered-kmer-size 14'`. This can be used with 30x or more of coverage, below that the defaults are OK.

My assembly continuity is not good, how can I improve it?

The most important determinant for assembly quality is sequence length, followed by the repeat complexity/heterozygosity of your sample. The first thing to check is the amount of corrected bases output by the correction step. This is logged in the stdout of Canu or in canu-scripts/canu.*.out if you are running in a grid environment. For example on a haploid H. sapiens sample:

```
-- BEGIN TRIMMING
--
...
-- In gatekeeper store 'chm1/trimming/asm.gkpStore':
--   Found 5459105 reads.
--   Found 91697412754 bases (29.57 times coverage).
...
```

Canu tries to correct the longest 40X of data. Some loss is normal but having output coverage below 20-25X is a sign that correction did not work well (assuming you have more input coverage than that). If that is the case, re-running with `corMhapSensitivity=normal` if you have >50X or `corMhapSensitivity=high corMinCoverage=0` otherwise can help. You can also increase the target coverage to correct `corOutCoverage=100` to get more correct sequences for assembly. If there are sufficient corrected reads, the poor assembly is likely due to either repeats in the genome being greater than read lengths or a high heterozygosity in the sample. Stay tuned for mor information on tuning unitigging in those instances.

What parameters can I tweak?

For all stages:

- `rawErrorRate` is the maximum expected difference in an alignment of two _uncorrected_ reads. It is a meta-parameter that sets other parameters.
- `correctedErrorRate` is the maximum expected difference in an alignment of two _corrected_ reads. It is a meta-parameter that sets other parameters. (If you're used to the `errorRate` parameter, multiply that by 3 and use it here.)
- `minReadLength` and `minOverlapLength`. The defaults are to discard reads shorter than 1000bp and to not look for overlaps shorter than 500bp. Increasing `minReadLength` can improve run time, and increasing `minOverlapLength` can improve assembly quality by removing false overlaps. However, increasing either too much will quickly degrade assemblies by either omitting valuable reads or missing true overlaps.

For correction:

- `corOutCoverage` controls how much coverage in corrected reads is generated. The default is to target 40X, but, for various reasons, this results in 30X to 35X of reads being generated.
- `corMinCoverage`, loosely, controls the quality of the corrected reads. It is the coverage in evidence reads that is needed before a (portion of a) corrected read is reported. Corrected reads are generated as a consensus of other reads; this is just the minimum ocverage needed for the consensus sequence to be reported. The default is based on input read coverage: 0x coverage for less than 30X input coverage, and 4x coverage for more than that.

For assembly:

- `utgOvlErrorRate` is essientially a speed optimization. Overlaps above this error rate are not computed. Setting it too high generally just wastes compute time, while setting it too low will degrade assemblies by missing true overlaps between lower quality reads.
- `utgGraphDeviation` and `utgRepeatDeviation` what quality of overlaps are used in contig construction or in breaking contigs at false repeat joins, respectively. Both are in terms of a deviation from the mean error rate in the longest overlaps.
- `utgRepeatConfusedBP` controls how similar a true overlap (between two reads in the same contig) and a false overlap (between two reads in different contigs) need to be before the contig is split. When this occurs, it isn't clear which overlap is 'true' - the longer one or the slightly shorter one - and the contig is split to avoid misassemblies.

For polyploid genomes:

Generally, there's a couple of ways of dealing with the ploidy.

1. **Avoid collapsing the genome** so you end up with double (assuming diploid) the genome size as long as your divergence is above about 2% (for PacBio data). Below this divergence, you'd end up collapsing the variations. We've used the following parameters for polyploid populations (PacBio data):

   ```
   corOutCoverage=200 correctedErrorRate=0.040 "batOptions=-dg
   3 -db 3 -dr 1 -ca 500 -cp 50"
   ```

   This will output more corrected reads (than the default 40x). The latter option will be more conservative at picking the error rate to use for the assembly to try to maintain haplotype separation. If it works, you'll end up with an assembly >= 2x your haploid genome size. Post-processing using gene information or other synteny information is required to remove redunancy from this assembly.

2. **Smash haplotypes together** and then do phasing using another approach (like HapCUT2 or whatshap or others). In that case you want to do the opposite, increase the error rates used for finding overlaps:

   ```
   corOutCoverage=200 ovlErrorRate=0.15 obtErrorRate=0.15
   ```

   Error rates for trimming (`obtErrorRate`) and assembling (`batErrorRate`) can usually be left as is. When trimming, reads will be trimmed using other reads in the same chromosome (and probably some reads from other chromosomes). When assembling, overlaps well outside the observed error rate distribution are discarded.

For low coverage:

- For less than 30X coverage, increase the alllowed difference in overlaps from 4.5% to 7.5% (or more) with `correctedErrorRate=0.075`, to adjust for inferior read correction. Canu will automatically reduce `corMinCoverage` to zero to correct as many reads as possible.

For high coverage:

- For more than 60X coverage, decrease the allowed difference in overlaps from 4.5% to 4.0% with `correctedErrorRate=0.040`, so that only the better corrected reads are used. This is primarily an optimization for speed and generally does not change assembly continuity.

My asm.contigs.fasta is empty, why?

Canu creates three assembled sequence *output files*: `<prefix>.contigs.fasta`, `<prefix>.unitigs.fasta`, and `<prefix>.unassembled.fasta`, where contigs are the primary output, unitigs are the primary output split at alternate paths, and unassembled are the leftover pieces.

The contigFilter parameter sets several parameters that control how small or low coverage initial contigs are handled. By default, initial contigs with more than 50% of the length at less than 5X coverage will be classified as 'unassembled' and removed from the assembly, that is, `contigFilter="2 0 1.0 0.5 5"`. The filtering can be disabled by changing the last number from '5' to '0' (meaning, filter if 50% is less than 0X coverage).

Why is my assembly is missing my favorite short plasmid?

Only the longest 40X of data (based on the specified genome size) is used for correction. Datasets with uneven coverage or small plasmids can fail to generate enough corrected reads to give enough coverage for assembly, resulting in gaps in the genome or even no reads for small plasmids. Set `corOutCoverage=1000` (or any value greater than your total input coverage) to correct all input data.

An alternate approach is to correct all reads (`-correct corOutCoverage=1000`) then assemble 40X of reads picked at random from the `<prefix>.correctedReads.fasta.gz` output.

Why do I get less corrected read data than I asked for?

Some reads are trimmed during correction due to being chimeric or because there wasn't enough evidence to generate a quality corrected sequence. Typically, this results in a 25% loss. Setting `corMinCoverage=0` will report all bases, even low those of low quality. Canu will trim these in its 'trimming' phase before assembly.

What is the minimum coverage required to run Canu?

For eukaryotic genomes, coverage more than 20X is enough to outperform current hybrid methods.

My genome is AT (or GC) rich, do I need to adjust parameters? What about highly repetitive genomes?

On bacterial genomes, no adjustment of parameters is (usually) needed. See the next question.

On repetitive genomes with with a significantly skewed AT/GC ratio, the Jaccard estimate used by MHAP is biased. Setting `corMaxEvidenceErate=0.15` is sufficient to correct for the bias in our testing.

In general, with high coverage repetitive genomes (such as plants) it can be beneficial to set the above parameter anyway, as it will eliminate repetitive matches, speed up the assembly, and sometime improve unitigs.

How can I send data to you?

FTP to ftp://ftp.cbcb.umd.edu/incoming/sergek. This is a write-only location that only the Canu developers can see.

Here is a quick walk-through using a command-line ftp client (should be available on most Linux and OSX installations). Say we want to transfer a file named `reads.fastq`. First, run `ftp ftp.cbcb.umd.edu`, specify `anonymous` as the user name and hit return for password (blank). Then:

That's it, you won't be able to see the file but we can download it.

CHAPTER 3

Canu Tutorial

Canu assembles reads from PacBio RS II or Oxford Nanopore MinION instruments into uniquely-assemblable contigs, unitigs. Canu owes lots of it design and code to celera-assembler.

Canu can be run using hardware of nearly any shape or size, anywhere from laptops to computational grids with thousands of nodes. Obviouisly, larger assemblies will take a long time to compute on laptops, and smaller assemblies can't take advantage of hundreds of nodes, so what is being assembled plays some part in determining what hardware can be effectively used.

Most algorithms in canu have been multi-threaded (to use all the cores on a single node), parallelized (to use all the nodes in a grid), or both (all the cores on all the nodes).

Canu, the command

The **canu** command is the 'executive' program that runs all modules of the assembler. It oversees each of the three top-level tasks (correction, trimming, unitig construction), each of which consists of many steps. Canu ensures that input files for each step exist, that each step successfully finished, and that the output for each step exists. It does minor bits of processing, such as reformatting files, but generally just executes other programs.

```
canu [-correct | -trim | -assemble | -trim-assemble] \
  [-s <assembly-specifications-file>] \
   -p <assembly-prefix> \
   -d <assembly-directory> \
   genomeSize=<number>[g|m|k] \
   [other-options] \
   [-pacbio-raw | -pacbio-corrected | -nanopore-raw | -nanopore-corrected] *fastq
```

The -p option, to set the file name prefix of intermediate and output files, is mandatory. If -d is not supplied, canu will run in the current directory, otherwise, Canu will create the *assembly-directory* and run in that directory. It is _not_ possible to run two different assemblies in the same directory.

The -s option will import a list of parameters from the supplied specification ('spec') file. These parameters will be applied before any from the command line are used, providing a method for setting commonly used parameters, but overriding them for specific assemblies.

By default, all three top-level tasks are performed. It is possible to run exactly one task by using the -correct, -trim or -assemble options. These options can be useful if you want to correct reads once and try many different assemblies. We do exactly that in the *Canu Quick Start*. Additionally, suppling pre-corrected reads with -pacbio-corrected or -nanopore-corrected will run only the trimming (-trim) and assembling (-assemble) stages.

Parameters are key=value pairs that configure the assembler. They set run time parameters (e.g., memory, threads, grid), algorithmic parameters (e.g., error rates, trimming aggressiveness), and enable or disable entire processing steps (e.g., don't correct errors, don't search for subreads). They are described later. One parameter is required: the genomeSize (in bases, with common SI prefixes allowed, for example, 4.7m or 2.8g; see genomeSize). Parameters are listed in the *Canu Parameter Reference*, but the common ones are described in this document.

Reads are supplied to canu by options that options that describe how the reads were generated, and what level of quality they are, for example, -pacbio-raw indicates the reads were generated on a PacBio RS II instrument, and have had no processing done to them. Each file of reads supplied this way becomes a 'library' of reads. The reads should have been (physically) generated all at the same time using the same steps, but perhaps sequenced in multiple batches. In canu, each library has a set of options setting various algorithmic parameters, for example, how aggressively to trim. To explicitly set library parameters, a text 'gkp' file describing the library and the input files must be created. Don't worry too much about this yet, it's an advanced feature, fully described in Section gkp-files.

The read-files contain sequence data in either FASTA or FASTQ format (or both! A quirk of the implementation allows files that contain both FASTA and FASTQ format reads). The files can be uncompressed, gzip, bzip2 or xz compressed. We've found that "gzip -1" provides good compression that is fast to both compress and decompress. For 'archival' purposes, we use "xz -9".

Canu, the pipeline

The canu pipeline, that is, what it actually computes, comprises of computing overlaps and processing the overlaps to some result. Each of the three tasks (read correction, read trimming and unitig construction) follow the same pattern:

- Load reads into the read database, gkpStore.
- Compute k-mer counts in preparation for the overlap computation.
- Compute overlaps.
- Load overlaps into the overlap database, ovlStore.
- Do something interesting with the reads and overlaps.
  - The read correction task will replace the original noisy read sequences with consensus sequences computed from overlapping reads.
  - The read trimming task will use overlapping reads to decide what regions of each read are high-quality sequence, and what regions should be trimmed. After trimming, the single largest high-quality chunk of sequence is retained.
  - The unitig construction task finds sets of overlaps that are consistent, and uses those to place reads into a multialignment layout. The layout is then used to generate a consensus sequence for the unitig.

Module Tags

Because each of the three tasks share common algorithms (all compute overlaps, two compute consensus sequences, etc), parameters are differentiated by a short prefix 'tag' string. This lets canu have one generic parameter that can be set to different values for each stage in each task. For example, "corOvlMemory" will set memory usage for overlaps being generated for read correction; "obtOvlMemory" for overlaps generated for Overlap Based Trimming; "utgOvlMemory" for overlaps generated for unitig construction.

The tags are:

Tag	Usage
master	the canu script itself, and any components that it runs directly
cns	unitig consensus generation
cor	read correction generation
red	read error detection
oea	overlap error adjustment
ovl	the standard overlapper
corovl	the standard overlapper, as used in the correction phase
obtovl	the standard overlapper, as used in the trimming phase
utgovl	the standard overlapper, as used in the assembly phase
mhap	the mhap overlapper
cormhap	the mhap overlapper, as used in the correction phase
obtmhap	the mhap overlapper, as used in the trimming phase
utgmhap	the mhap overlapper, as used in the assembly phase
mmap	the minimap overlapper
cormmap	the minimap overlapper, as used in the correction phase
obtmmap	the minimap overlapper, as used in the trimming phase
utgmmap	the minimap overlapper, as used in the assembly phase
ovb	the bucketizing phase of overlap store building
ovs	the sort phase of overlap store building

We'll get to the details eventually.

Execution Configuration

There are two modes that canu runs in: locally, using just one machine, or grid-enabled, using multiple hosts managed by a grid engine. LSF, PBS/Torque, PBSPro, Sun Grid Engine (and derivations), and Slurm are supported, though LSF has has limited testing. Section *Grid Engine Configuration* has a few hints on how to set up a new grid engine.

By default, if a grid is detected the canu pipeline will immediately submit itself to the grid and run entirely under grid control. If no grid is detected, or if option `useGrid=false` is set, canu will run on the local machine.

In both cases, Canu will auto-detect available resources and configure job sizes based on the resources and genome size you're assembling. Thus, most users should be able to run the command without modifying the defaults. Some advanced options are outlined below. Each stage has the same five configuration options, and tags are used to specialize the option to a specific stage. The options are:

useGrid<tag>=boolean Run this stage on the grid, usually in parallel.

gridOptions<tag>=string Supply this string to the grid submit command.

<tag>Memory=integer Use this many gigabytes of memory, per process.

<tag>Threads Use this many compute threads per process.

<tag>Concurrency If not on the grid, run this many jobs at the same time.

Global grid options, applied to every job submitted to the grid, can be set with 'gridOptions'. This can be used to add accounting information or access credentials.

A name can be associated with this compute using 'gridOptionsJobName'. Canu will work just fine with no name set, but if multiple canu assemblies are running at the same time, they will tend to wait for each others jobs to finish. For example, if two assemblies are running, at some point both will have overlap jobs running. Each assembly will be waiting for all jobs named 'ovl_asm' to finish. Had the assemblies specified job names, gridOptionsJobName=apple and gridOptionsJobName=orange, then one would be waiting for jobs named 'ovl_asm_apple', and the other would be waiting for jobs named 'ovl_asm_orange'.

Error Rates

Canu expects all error rates to be reported as fraction error, not as percent error. We're not sure exactly why this is so. Previously, it used a mix of fraction error and percent error (or both!), and was a little confusing. Here's a handy table you can print out that converts between fraction error and percent error. Not all values are shown (it'd be quite a large table) but we have every confidence you can figure out the missing values:

Fraction Error	Percent Error
0.01	1%
0.02	2%
0.03	3%
.	.
.	.
0.12	12%
.	.
.	.

Canu error rates always refer to the percent difference in an alignment of two reads, not the percent error in a single read, and not the amount of variation in your reads. These error rates are used in two different ways: they are used to limit what overlaps are generated, e.g., don't compute overlaps that have more than 5% difference; and they are used to tell algorithms what overlaps to use, e.g., even though overlaps were computed to 5% difference, don't trust any above 3% difference.

There are seven error rates. Three error rates control overlap creation (*corOvlErrorRate*, *obtOvlErrorRate* and *utgOvlErrorRate*), and four error rates control algorithms (corErrorRate, *obtErrorRate*, *utgErrorRate*, *cnsErrorRate*).

The three error rates for overlap creation apply to the *ovl* overlap algorithm and the *mhapReAlign* option used to generate alignments from *mhap* or *minimap* overlaps. Since *mhap* is used for generating correction overlaps, the *corOvlErrorRate* parameter is not used by default. Overlaps for trimming and assembling use the *ovl* algorithm, therefore, *obtOvlErrorRate* and *utgOvlErrorRate* are used.

The four algoriothm error rates are used to select which overlaps can be used for correcting reads (corErrorRate); which overlaps can be used for trimming reads (*obtErrorRate*); which overlaps can be used for assembling reads (*utgErrorRate*). The last error rate, *cnsErrorRate*, tells the consensus algorithm to not trust read alignments above that value.

For convenience, two meta options set the error rates used with uncorrected reads (*rawErrorRate*) or used with corrected reads. (*correctedErrorRate*). The default depends on the type of read being assembled.

Parameter	PacBio	Nanopore
rawErrorRate	0.300	0.500
correctedErrorRate	0.045	0.144

In practice, only *correctedErrorRate* is usually changed. The *Canu FAQ* has *specific suggestions* on when to change this.

Canu v1.4 and earlier used the *errorRate* parameter, which set the expected rate of error in a single corrected read.

Minimum Lengths

Two minimum sizes are known:

minReadLength Discard reads shorter than this when loading into the assembler, and when trimming reads.

minOverlapLength Do not save overlaps shorter than this.

Overlap configuration

The largest compute of the assembler is also the most complicated to configure. As shown in the 'module tags' section, there are up to eight (!) different overlapper configurations. For each overlapper ('ovl' or 'mhap') there is a global configuration, and three specializations that apply to each stage in the pipeline (correction, trimming or assembly).

Like with 'grid configuration', overlap configuration uses a 'tag' prefix applied to each option. The tags in this instance are 'cor', 'obt' and 'utg'.

For example:

- To change the k-mer size for all instances of the ovl overlapper, 'merSize=23' would be used.
- To change the k-mer size for just the ovl overlapper used during correction, 'corMerSize=16' would be used.
- To change the mhap k-mer size for all instances, 'mhapMerSize=18' would be used.
- To change the mhap k-mer size just during correction, 'corMhapMerSize=15' would be used.
- To use minimap for overlap computation just during correction, 'corOverlapper=minimap' would be used.

Ovl Overlapper Configuration

<tag>Overlapper select the overlap algorithm to use, 'ovl' or 'mhap'.

Ovl Overlapper Parameters

<tag>ovlHashBlockLength how many bases to reads to include in the hash table; directly controls process size

<tag>ovlRefBlockSize how many reads to compute overlaps for in one process; directly controls process time

<tag>ovlRefBlockLength same, but use 'bases in reads' instead of 'number of reads'

<tag>ovlHashBits size of the hash table (SHOULD BE REMOVED AND COMPUTED, MAYBE TWO PASS)

<tag>ovlHashLoad how much to fill the hash table before computing overlaps (SHOULD BE REMOVED)

<tag>ovlMerSize size of kmer seed; smaller - more sensitive, but slower

The overlapper will not use frequent kmers to seed overlaps. These are computed by the 'meryl' program, and can be selected in one of three ways.

Terminology. A k-mer is a contiguous sequence of k bases. The read 'ACTTA' has two 4-mers: ACTT and CTTA. To account for reverse-complement sequence, a 'canonical kmer' is the lexicographically smaller of the forward and reverse-complemented kmer sequence. Kmer ACTT, with reverse complement AAGT, has a canonical kmer AAGT. Kmer CTTA, reverse-complement TAAG, has canonical kmer CTTA.

A 'distinct' kmer is the kmer sequence with no count associated with it. A 'total' kmer (for lack of a better term) is the kmer with its count. The sequence TCGTTTTTTTCGTCG has 12 'total' 4-mers and 8 'distinct' kmers.

```
TCGTTTTTTTCGTCG  count
TCGT             2 distinct-1
 CGTT            1 distinct-2
  GTTT           1 distinct-3
   TTTT          4 distinct-4
    TTTT         4 copy of distinct-4
     TTTT        4 copy of distinct-4
      TTTT       4 copy of distinct-4
       TTTC      1 distinct-5
        TTCG     1 distinct-6
         TCGT    2 copy of distinct-1
          CGTC   1 distinct-7
           GTCG  1 distinct-8
```

<tag>MerThreshold any kmer with count higher than N is not used

<tag>MerDistinct pick a threshold so as to seed overlaps using this fraction of all distinct kmers in the input. In the example above, fraction 0.875 of the k-mers (7/8) will be at or below threshold 2.

<tag>MerTotal pick a threshold so as to seed overlaps using this fraction of all kmers in the input. In the example above, fraction 0.667 of the k-mers (8/12) will be at or below threshold 2.

<tag>FrequentMers don't compute frequent kmers, use those listed in this fasta file

Mhap Overlapper Parameters

<tag>MhapBlockSize Chunk of reads that can fit into 1GB of memory. Combined with memory to compute the size of chunk the reads are split into.

<tag>MhapMerSize Use k-mers of this size for detecting overlaps.

<tag>ReAlign After computing overlaps with mhap, compute a sequence alignment for each overlap.

<tag>MhapSensitivity Either 'normal', 'high', or 'fast'.

Mhap also will down-weight frequent kmers (using tf-idf), but it's selection of frequent is not exposed.

Minimap Overlapper Parameters

<tag>MMapBlockSize Chunk of reads that can fit into 1GB of memory. Combined with memory to compute the size of chunk the reads are split into.

<tag>MMapMerSize Use k-mers of this size for detecting overlaps

Minimap also will ignore high-frequency minimzers, but it's selection of frequent is not exposed.

Outputs

As Canu runs, it outputs status messages, execution logs, and some analysis to the console. Most of the analysis is captured in `<prefix>.report` as well.

LOGGING

<prefix>.report Most of the analysis reported during assembly.

READS

<prefix>.correctedReads.fasta.gz The reads after correction.

<prefix>.trimmedReads.fasta.gz The corrected reads after overlap based trimming.

SEQUENCE

<prefix>.contigs.fasta Everything which could be assembled and is part of the primary assembly, including both unique and repetitive elements.

<prefix>.unitigs.fasta Contigs, split at alternate paths in the graph.

<prefix>.unassembled.fasta Reads and low-coverage contigs which could not be incorporated into the primary assembly.

The header line for each sequence provides some metadata on the sequence.:

```
>tig######## len=<integer> reads=<integer> covStat=<float> gappedBases=<yes|no> class=
↪<contig|bubble|unassm> suggestRepeat=<yes|no> suggestCircular=<yes|no>

len
   Length of the sequence, in bp.

reads
   Number of reads used to form the contig.

covStat
   The log of the ratio of the contig being unique versus being two-copy, based on␣
↪the read arrival rate.  Positive values indicate more likely to be unique, while␣
↪negative values indicate more likely to be repetitive.  See `Footnote 24 <http://
↪science.sciencemag.org/content/287/5461/2196.full#ref-24>`_ in `Myers et al., A␣
↪Whole-Genome Assembly of Drosophila <http://science.sciencemag.org/content/287/5461/
↪2196.full>`_.

gappedBases
   If yes, the sequence includes all gaps in the multialignment.

class
   Type of sequence.  Unassembled sequences are primarily low-coverage sequences␣
↪spanned by a single read.

suggestRepeat
   If yes, sequence was detected as a repeat based on graph topology or read overlaps␣
↪to other sequences.

suggestCircular
   If yes, sequence is likely circular.  Not implemented.
```

GRAPHS

<prefix>.contigs.gfa Unused or ambiguous edges between contig sequences. Bubble edges cannot be represented in this format.

<prefix>.unitigs.gfa Contigs split at bubble intersections.

<prefix>.unitigs.bed The position of each unitig in a contig.

METADATA

The layout provides information on where each read ended up in the final assembly, including contig and positions. It also includes the consensus sequence for each contig.

<prefix>.contigs.layout, <prefix>.unitigs.layout (undocumented)

<prefix>.contigs.layout.readToTig, <prefix>.unitigs.layout.readToTig The position of each read in a contig (unitig).

<prefix>.contigs.layout.tigInfo, <prefix>.unitigs.layout.tigInfo A list of the contigs (unitigs), lengths, coverage, number of reads and other metadata. Essentially the same information provided in the FASTA header line.

CHAPTER 4

Canu Pipeline

The pipeline is described in Koren S, Walenz BP, Berlin K, Miller JR, Phillippy AM. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. bioRxiv. (2016). Figure 1 of the paper shows the primary pipeline (below, top) and the supplement contains the sub-pipeline for building read and overlap databases (below, bottom).

CHAPTER 5

Canu Parameter Reference

To get the most up-to-date options, run

    canu -options

The default values below will vary based on the input data type and genome size.

Boolean options accept true/false or 1/0.

Memory sizes are assumed to be in gigabytes if no units are supplied. Values may be non-integer with or without a unit - 'k' for kilobytes, 'm' for megabytes, 'g' for gigabytes or 't' for terabytes. For example, "0.25t" is equivalent to "256g" (or simply "256").

Global Options

The catch all category.

errorRate <float=unset> (OBSOLETE) This parameter was removed on January 27th, 2016, and is valid only in Canu 1.4 or earlier. Canu currently still accepts the *errorRate* parameter, but its use is strongly discouraged.

The expected error in a single corrected read. The seven error rates were then set to three times this value (except for corErrorRate).

rawErrorRate <float=unset> The allowed difference in an overlap between two uncorrected reads, expressed as fraction error. Sets corOvlErrorRate and corErrorRate. The *rawErrorRate* typically does not need to be modified. It might need to be increased if very early reads are being assembled. The default is 0.300 For PacBio reads, and 0.500 for Nanopore reads.

correctedErrorRate <float=unset> The allowed difference in an overlap between two corrected reads, expressed as fraction error. Sets obtOvlErrorRate, utgOvlErrorRate, obtErrorRate, utgErrorRate, and cnsErrorRate. The *correctedErrorRate* can be adjusted to account for the quality of read correction, for the amount of divergence in the sample being assembled, and for the amount of sequence being assembled. The default is 0.045 for PacBio reads, and 0.144 for Nanopore reads.

For low coverage datasets (less than 30X), we recommend increasing *correctedErrorRate* slightly, by 1% or so.

For high-coverage datasets (more than 60X), we recommend decreasing *correctedErrorRate* slighly, by 1% or so.

Raising the *correctedErrorRate* will increase run time. Likewise, decreasing *correctedErrorRate* will decrease run time, at the risk of missing overlaps and fracturing the assembly.

minReadLength <integer=1000> Reads shorter than this are not loaded into the assembler. Reads output by correction and trimming that are shorter than this are discarded.

Must be no smaller than minOverlapLength.

If set high enough, the gatekeeper module will halt as too many of the input reads have been discarded. Set *stopOnReadQuality* to false to avoid this.

minOverlapLength <integer=500> Overlaps shorter than this will not be discovered. Smaller values can be used to overcome lack of read coverage, but will also lead to false overlaps and potential misassemblies. Larger values will result in more correct assemblies, but more fragmented, assemblies.

Must be no bigger than minReadLength.

genomeSize <float=unset> ***required*** An estimate of the size of the genome. Common suffices are allowed, for example, 3.7m or 2.8g.

The genome size estimate is used to decide how many reads to correct (via the *corOutCoverage* parameter) and how sensitive the mhap overlapper should be (via the *mhapSensitivity* parameter). It also impacts some logging, in particular, reports of NG50 sizes.

canuIteration <internal parameter, do not use> Which parallel iteration is being attempted.

canuIterationMax <integer=2> How many parallel iterations to try. Ideally, the parallel jobs, run under grid control, would all finish successfully on the first try. Sometimes, jobs fail due to other jobs exhausting resources (memory), or by the node itself failing. In this case, canu will launch the jobs again. This parameter controls how many times it tries.

onSuccess <string=unset> Execute the command supplied when Canu successfully completes an assembly. The command will execute in the <assembly-directory> (the -d option to canu) and will be supplied with the name of the assembly (the -p option to canu) as its first and only parameter.

onFailure <string=unset> Execute the command supplied when Canu terminates abnormally. The command will execute in the <assembly-directory> (the -d option to canu) and will be supplied with the name of the assembly (the -p option to canu) as its first and only parameter.

There are two exceptions when the command is not executed: if a 'spec' file cannot be read, or if canu tries to access an invalid parameter. The former will be reported as a command line error, and canu will never start. The latter should never occur except when developers are developing the software.

Process Control

showNext <boolean=false> Report the first major command that would be run, but don't run it. Processing to get to that command, for example, checking the output of the previous command or preparing inputs for the next command, is still performed.

stopAfter <string=undefined> If set, Canu will stop processing after a specific stage in the pipeline finishes.

Valid values for `stopAfter` are:

- `gatekeeper` - stops after the reads are loaded into the assembler read database.
- `meryl` - stops after frequent kmers are tabulated.
- `overlapConfigure` - stops after overlap jobs are configured.

- `overlap` - stops after overlaps are generated, before they are loaded into the overlap database.
- `overlapStoreConfigure` - stops after the `ovsMethod=parallel` jobs are configured; has no impact for `ovsMethod=sequential`.
- `overlapStore` - stops after overlaps are loaded into the overlap database.
- `readCorrection` - stops after corrected reads are generated.
- `readTrimming` - stops after trimmed reads are generated.
- `unitig` - stops after unitigs and contigs are created.
- `consensusConfigure` - stops after consensus jobs are configured.
- `consensus` - stops after consensus sequences are loaded into the databases.

General Options

pathMap <string=undefined> A text file containing lines that map a hostname to a path to the assembler binaries. This can be used to provide fine-grained binary directories, for example, when incompatible versions of the operating system are used, or when canu is installed in a non-standard way.

The hostname must be the same as returned by 'uname -n'. For example:

```
grid01   /grid/software/canu/Linux-amd64/bin/
devel01  /devel/canu/Linux-amd64/bin/
```

shell <string="/bin/sh"> A path to a Bourne shell, to be used for executing scripts. By default, '/bin/sh', which is typically the same as 'bash'. C shells (csh, tcsh) are not supported.

java <string="java"> A path to a Java application launcher of at least version 1.8.

gnuplot <string="gnuplot"> A path to the gnuplot graphing utility.

gnuplotImageFormat <string="png"> The type of image to generate in gnuplot. By default, canu will use png, svg or gif, in that order.

gnuplotTested <boolean=false> If set, skip the tests to determine if gnuplot will run, and to decide the image type to generate. This is used when gnuplot fails to run, or isn't even installed, and allows canu to continue execution without generating graphs.

File Staging

The correction stage of Canu requires random access to all the reads. Performance is greatly improved if the gkpStore database of reads is copied locally to each node that computes corrected read consensus sequences. This 'staging' is enabled by supplying a path name to fast local storage with the *stageDirectory* option, and, optionally, requesting access to that resource from the grid with the *gridEngineStageOption* option.

stageDirectory <string=undefined> A path to a directory local to each compute node. The directory should use an environment variable specific to the grid engine to ensure that it is unique to each task.

For example, in Sun Grid Engine, */scratch/JOB_ID-SGE_TASK_ID* will use both the numeric job ID and the numeric task ID. In SLURM, */scratch/$SLRUM_JOBID* accomplishes the same.

If specified on the command line, be sure to escape the dollar sign, otherwise the shell will try to expand it before Canu sees the option: *stageDirectory=/scratch/JOB_ID-SGE_TASK_ID*.

If specified in a specFile, do not escape the dollar signs.

gridEngineStageOption <string=undefined> This string is passed to the job submission command, and is expected to request local disk space on each node. It is highly grid specific. The string *DISK_SPACE* will be replaced with the amount of disk space needed, in gigabytes.

On SLURM, an example is *–gres=lscratch:DISK_SPACE*

Cleanup Options

saveOverlaps <boolean=false> If set, do not remove raw overlap output from either mhap or overlapInCore. Normally, this output is removed once the overlaps are loaded into an overlap store.

saveReadCorrections <boolean=false. If set, do not remove raw corrected read output from correction/2-correction. Normally, this output is removed once the corrected reads are generated.

saveIntermediates <boolean=false> If set, do not remove intermediate outputs. Normally, intermediate files are removed once they are no longer needed.

NOT IMPLEMENTED.

saveMerCounts <boolean=false> If set, do not remove meryl binary databases.

Overlapper Configuration

Overlaps are generated for three purposes: read correction, read trimming and unitig construction. The algorithm and parameters used can be set independently for each set of overlaps.

Two overlap algorithms are in use. One, mhap, is typically applied to raw uncorrected reads and returns alignment-free overlaps with imprecise extents. The other, the original overlapper algorithm 'ovl', returns alignments but is much more expensive.

There are three sets of parameters, one for the 'mhap' algorithm, one for the 'ovl' algorithm, and one for the 'minimap' algorithm. Parameters used for a specific type of overlap are set by a prefix on the option: 'cor' for read correction, 'obt' for read trimming ('overlap based trimming') or 'utg' for unitig construction. For example, 'corOverlapper=ovl' would set the overlapper used for read correction to the 'ovl' algorithm.

{prefix}Overlapper <string=see-below> Specify which overlap algorith, 'mhap' or 'ovl' or 'minimap'. The default is to use 'mhap' for 'cor' and 'ovl' for both 'obt' and 'utg'.

Overlapper Configuration, ovl Algorithm

{prefix}OvlErrorRate <float=unset> Overlaps above this error rate are not computed. * *corOvlErrorRate* applies to overlaps generated for correcting reads; * *obtOvlErrorRate* applied to overlaps generated for trimming reads; * *utgOvlErrorRate* applies to overlaps generated for assembling reads. These limits apply to the 'ovl' overlap algorithm and when alignments are computed for mhap overlaps with *mhapReAlign*.

{prefix}OvlFrequentMers <string=undefined> Do not seed overlaps with these kmers (fasta format).

{prefix}OvlHashBits <integer=unset> Width of the kmer hash. Width 22=1gb, 23=2gb, 24=4gb, 25=8gb. Plus 10b per corOvlHashBlockLength.

{prefix}OvlHashBlockLength <integer=unset> Amount of sequence (bp to load into the overlap hash table.

{prefix}OvlHashLoad <integer=unset> Maximum hash table load. If set too high, table lookups are inefficent; if too low, search overhead dominates run time.

{prefix}OvlMerDistinct <integer=unset> K-mer frequency threshold; the least frequent fraction of distinct mers can seed overlaps.

{prefix}OvlMerSize <integer=unset> K-mer size for seeds in overlaps.

{prefix}OvlMerThreshold <integer=unset> K-mer frequency threshold; mers more frequent than this count are not used to seed overlaps.

{prefix}OvlMerTotal <integer=unset> K-mer frequency threshold; the least frequent fraction of all mers can seed overlaps.

{prefix}OvlRefBlockLength <integer=unset> Amount of sequence (bp to search against the hash table per batch.

{prefix}OvlRefBlockSize <integer=unset> Number of reads to search against the hash table per batch.

Overlapper Configuration, mhap Algorithm

{prefix}MhapBlockSize <integer=unset> Number of reads per 1GB block. Memory * size is loaded into memory per job.

{prefix}MhapMerSize <integer=unset> K-mer size for seeds in mhap.

{prefix}ReAlign <boolean=false> Compute actual alignments from mhap overlaps. uses either obtErrorRate or ovlErrorRate, depending on which overlaps are computed)

{prefix}MhapSensitivity <string="normal"> Coarse sensitivity level: 'low', 'normal' or 'high'. Based on read coverage (which is impacted by genomeSize), 'low' sensitivity is used if coverage is more than 60; 'normal' is used if coverage is between 60 and 30, and 'high' is used for coverages less than 30.

Overlapper Configuration, mhap Algorithm

{prefix}MMapBlockSize <integer=unset> Number of reads per 1GB block. Memory * size is loaded into memory per job.

{prefix}MMapMerSize <integer=unset> K-mer size for seeds in minimap.

Overlap Store

The overlap algorithms return overlaps in an arbitrary order. The correction, trimming and assembly algorithms usually need to know all overlaps for a single read. The overlap store duplicates each overlap, sorts them by the first ID, and stores them for quick retrieval of all overlaps for a single read.

ovsMemory <float> How much memory, in gigabytes, to use for constructing overlap stores. Must be at least 256m or 0.25g.

ovsMethod <string="sequential"> Two construction algorithms are supported. One uses a single data stream, and is faster for small and moderate size assemblies. The other uses parallel data streams and can be faster (depending on your network disk bandwitdh) for moderate and large assemblies.

Meryl

The 'meryl' algorithm counts the occurrences of kmers in the input reads. It outputs a FASTA format list of frequent kmers, and (optionally) a binary database of the counts for each kmer in the input.

Meryl can run in (almost) any memory size, by splitting the computation into smaller (or larger) chunks.

merylMemory <integer=unset> Amount of memory, in gigabytes, to use for counting kmers.

merylThreads <integer=unset> Number of compute threads to use for kmer counting.

Overlap Based Trimming

obtErrorRate <float=unset> Stringency of overlaps to use for trimming reads.

trimReadsOverlap <integer=1> Minimum overlap between evidence to make contiguous trim.

trimReadsCoverage <integer=1> Minimum depth of evidence to retain bases.

Grid Engine Support

Canu directly supports most common grid scheduling systems. Under normal use, Canu will query the system for grid support, congigure itself for the machines available in the grid, then submit itself to the grid for execution. The Canu pipeline is a series of about a dozen steps that alternate between embarassingly parallel computations (e.g., overlap computation) and sequential bookkeeping steps (e.g., checking if all overlap jobs finished). This is entirely managed by Canu.

Canu has first class support for the various schedulers derived from Sun Grid Engine (Univa, Son of Grid Engine) and the Simple Linux Utility for Resource Management (SLURM), meaning that the devlopers have direct access to these systems. Platform Computing's Load Sharing Facility (LSF) and the various schedulers derived from the Portable Batch System (PBS, Torque and PBSPro) are supported as well, but without developer access bugs do creep in. As of Canu v1.5, support seems stable and working.

useGrid <boolean=true> Master control. If 'false', no algorithms will run under grid control. Does not change the value of the other useGrid options.

If 'remote', jobs are configured for grid execution, but not submitted. A message, with commands to launch the job, is reported and canu halts execution.

Note that the host used to run canu for 'remote' execution must know about the grid, that is, it must be able to submit jobs to the grid.

It is also possible to enable/disable grid support for individual algorithms with options such as *useGridBAT*, *useGridCNS*, et cetera. This has been useful in the (far) past to prevent certain algorithms, notably overlap error adjustment, from running too many jobs concurrently and thrashing disk. Recent storage systems seem to be able to handle the load better – computers have gotten faster quicker than genomes have gotten larger.

There are many options for configuring a new grid ('gridEngine*') and for configuring how canu configures its computes to run under grid control ('gridOptions*'). The grid engine to use is specified with the 'gridEngine' option.

gridEngine <string> Which grid engine to use. Auto-detected. Possible choices are 'sge', 'pbs', 'pbspro', 'lsf' or 'slurm'.

Grid Engine Configuration

There are many options to configure support for a new grid engine, and we don't describe them fully. If you feel the need to add support for a new engine, please contact us. That said, file `src/pipeline/canu/Defaults.pm` lists a whole slew of parameters that are used to build up grid commands, they all start with `gridEngine`. For each grid, these parameters are defined in the various `src/pipeline/Grid_*.pm` modules. The parameters are used in `src/pipeline/canu/Execution.pm`.

For SGE grids, two options are sometimes necessary to tell canu about pecularities of your grid: `gridEngineThreadsOption` describes how to request multiple cores, and `gridEngineMemoryOption` describes how to request memory. Usually, canu can figure out how to do this, but sometimes it reports an error such as:

```
-- WARNING:  Couldn't determine the SGE parallel environment to run multi-threaded␣
↪codes.
--           Valid choices are (pick one and supply it to canu):
--             gridEngineThreadsOption="-pe make THREADS"
--             gridEngineThreadsOption="-pe make-dedicated THREADS"
--             gridEngineThreadsOption="-pe mpich-rr THREADS"
--             gridEngineThreadsOption="-pe openmpi-fill THREADS"
--             gridEngineThreadsOption="-pe smp THREADS"
--             gridEngineThreadsOption="-pe thread THREADS"
```

or:

```
-- WARNING:  Couldn't determine the SGE resource to request memory.
--           Valid choices are (pick one and supply it to canu):
--             gridEngineMemoryOption="-l h_vmem=MEMORY"
--             gridEngineMemoryOption="-l mem_free=MEMORY"
```

If you get such a message, just add the appropriate line to your canu command line. Both options will replace the uppercase text (THREADS or MEMORY) with the value canu wants when the job is submitted. For `gridEngineMemoryOption`, any number of `-l` options can be supplied; we could use `gridEngineMemoryOption="-l h_vmem=MEMORY -l mem_free=MEMORY"` to request both `h_vmem` and `mem_free` memory.

Grid Options

To run on the grid, each stage needs to be configured - to tell the grid how many cores it will use and how much memory it needs. Some support for this is automagic (for example, overlapInCore and mhap know how to do this), others need to be manually configured. Yes, it's a problem, and yes, we want to fix it.

The gridOptions* parameters supply grid-specific opitons to the grid submission command.

gridOptions <string=unset> Grid submission command options applied to all grid jobs

gridOptionsJobName <string=unset> Grid submission command jobs name suffix

gridOptionsBAT <string=unset> Grid submission command options applied to unitig construction with the bogart algorithm

gridOptionsGFA <string=unset> Grid submission command options applied to gfa alignment and processing

gridOptionsCNS <string=unset> Grid submission command options applied to unitig consensus jobs

gridOptionsCOR <string=unset> Grid submission command options applied to read correction jobs

gridOptionsExecutive <string=unset> Grid submission command options applied to master script jobs

gridOptionsOEA <string=unset> Grid submission command options applied to overlap error adjustment jobs

gridOptionsRED <string=unset> Grid submission command options applied to read error detection jobs

gridOptionsOVB <string=unset> Grid submission command options applied to overlap store bucketizing jobs

gridOptionsOVS <string=unset> Grid submission command options applied to overlap store sorting jobs

gridOptionsCORMHAP <string=unset> Grid submission command options applied to mhap overlaps for correction jobs

gridOptionsCOROVL <string=unset> Grid submission command options applied to overlaps for correction jobs

gridOptionsOBTMHAP <string=unset> Grid submission command options applied to mhap overlaps for trimming jobs

gridOptionsOBTOVL <string=unset> Grid submission command options applied to overlaps for trimming jobs

gridOptionsUTGMHAP <string=unset> Grid submission command options applied to mhap overlaps for unitig construction jobs

gridOptionsUTGOVL <string=unset> Grid submission command options applied to overlaps for unitig construction jobs

Algorithm Selection

Several algorithmic components of canu can be disabled, based on the type of the reads being assebmled, the type of processing desired, or the amount of comput resources available. Overlap

enableOEA <boolean=true> Do overlap error adjustment - comprises two steps: read error detection (RED and overlap error adjustment (OEA

Algorithm Execution Method

Canu has a fairly sophisticated (or complicated, depending on if it is working or not) method for dividing large computes, such as read overlapping and consensus, into many smaller pieces and then running those pieces on a grid or in parallel on the local machine. The size of each piece is generally determined by the amount of memory the task is allowed to use, and this memory size – actually a range of memory sizes – is set based on the genomeSize parameter, but can be set explicitly by the user. The same holds for the number of processors each task can use. For example, a genomeSize=5m would result in overlaps using between 4gb and 8gb of memory, and between 1 and 8 processors.

Given these requirements, Canu will pick a specific memory size and number of processors so that the maximum number of jobs will run at the same time. In the overlapper example, if we are running on a machine with 32gb memory and 8 processors, it is not possible to run 8 concurrent jobs that each require 8gb memory, but it is possible to run 4 concurrent jobs each using 6gb memory and 2 processors.

To completely specify how Canu runs algorithms, one needs to specify a maximum memory size, a maximum number of processors, and how many pieces to run at one time. Users can set these manually through the {prefix}Memory, {prefix}Threads and {prefix}Concurrency options. If they are not set, defaults are chosen based on genomeSize.

{prefix}Concurrency <integer=unset> Set the number of tasks that can run at the same time, when running without grid support.

{prefix}Threads <integer=unset> Set the number of compute threads used per task.

{prefix}Memory <integer=unset> Set the amount of memory, in gigabytes, to use for each job in a task.

Available prefixes are:

Prefix		Algorithm
	mhap	
cor		Overlap generation using the‘mhap’ algorithm for‘cor’=
	mmap	
cor		Overlap generation using the‘minimap’ algorithm for‘c
obt		
	ovl	
cor		Overlap generation using the‘overlapInCore’ algorithm
obt	ovb	Parallel overlap store bucketizing
utg		
	ovs	Parallel overlap store bucket sorting
	cor	Read correction
obt	red	Error detection in reads
utg	oea	Error adjustment in overlaps
	bat	Unitig/contig construction
	cns	Unitig/contig consensus

For example, ‘mhapMemory‘ would set the memory limit for computing overlaps with the mhap algorithm; ‘cormhapMemory’ would set the memory limit only when mhap is used for generating overlaps used for correction.

The ‘minMemory’, ‘maxMemory’, ‘minThreads’ and ‘maxThreads’ options will apply to all jobs, and can be used to artifically limit canu to a portion of the current machine. In the overlapper example above, setting maxThreads=4 would result in two concurrent jobs instead of four.

Overlap Error Adjustment

red = Read Error Detection oea = Overlap Error Adjustment

oeaBatchLength <unset> Number of bases per overlap error correction batch

oeaBatchSize <unset> Number of reads per overlap error correction batch

redBatchLength <unset> Number of bases per fragment error detection batch

redBatchSize <unset> Number of reads per fragment error detection batch

Unitigger

unitigger <string=”bogart”> Which unitig construction algorithm to use. Only “bogart” is supported.

utgErrorRate <float=unset> Stringency of overlaps used for constructing contigs. The *bogart* algorithm uses the distribution of overlap error rates to filter high error overlaps; *bogart* will never see overlaps with error higher than this parameter.

batOptions <unset> Advanced options to bogart

Consensus Partitioning

STILL DONE BY UNITIGGER, NEED TO MOVE OUTSIDE

cnsConsensus Which algorithm to use for computing consensus sequences. Only 'utgcns' is supported.

cnsPartitions Compute conseus by splitting the tigs into N partitions.

cnsPartitionMin Don't make a paritition with fewer than N reads

cnsMaxCoverage Limit unitig consensus to at most this coverage.

cnsErrorRate Inform the consensus genration algorithm of the amount of difference it should expect in a read-to-read alignment. Typically set to *utgOvlErrorRate*. If set too high, reads could be placed in an incorrect location, leading to errors in the consensus sequence. If set too low, reads could be omitted from the consensus graph (or multialignment, depending on algorithm), resulting in truncated consensus sequences.

Read Correction

The first step in Canu is to find high-error overlaps and generate corrected sequences for subsequent assembly. This is currently the fastest step in Canu. By default, only the longest 40X of data (based on the specified genome size) is used for correction. Typically, some reads are trimmed during correction due to being chimeric or having erroneous sequence, resulting in a loss of 20-25% (30X output). You can force correction to be non-lossy by setting *corMinCoverage=0*, in which case the corrected reads output will be the same length as the input data, keeping any high-error unsupported bases. Canu will trim these in downstream steps before assembly.

If you have a dataset with uneven coverage or small plasmids, correcting the longest 40X may not give you sufficient coverage of your genome/plasmid. In these cases, you can set *corOutCoverage=999*, or any value greater than your total input coverage which will correct and assemble all input data, at the expense of runtime.

corErrorRate <integer=unset> Do not use overlaps with error rate higher than this (estimated error rate for *mhap* and *minimap* overlaps).

corConsensus <string="falconpipe"> Which algorithm to use for computing read consensus sequences. Only 'falcon' and 'falconpipe' are supported.

corPartitions <integer=128> Partition read correction into N jobs

corPartitionMin <integer=25000> Don't make a read correction partition with fewer than N reads

corMinEvidenceLength <integer=unset> Limit read correction to only overlaps longer than this; default: unlimited

corMinCoverage <integer=4> Limit read correction to regions with at least this minimum coverage. Split reads when coverage drops below threshold.

corMaxEvidenceErate <integer=unset> Limit read correction to only overlaps at or below this fraction error; default: unlimited

corMaxEvidenceCoverageGlobal <string="1.0x"> Limit reads used for correction to supporting at most this coverage; default: 1.0 * estimated coverage

corMaxEvidenceCoverageLocal <string="2.0x"> Limit reads being corrected to at most this much evidence coverage; default: 10 * estimated coverage

corOutCoverage <integer=40> Only correct the longest reads up to this coverage; default 40

corFilter <string="expensive"> Method to filter short reads from correction; 'quick' or 'expensive' or 'none'

Output Filtering

contigFilter <minReads, integer=2> <minLength, integer=0> <singleReadSpan, float=1.0> <lowCovSpan, float=0.5> <lowCovD

Remove spurious assemblies from consideration. Any contig that meeds any of the following conditions is flagged as 'unassembled' and removed from further consideration:

- fewer than minReads reads
- shorter than minLength bases
- a single read covers more than singleReadSpan fraction of the contig
- more than lowCovSpan fraction of the contig is at coverage below lowCovDepth

This filtering is done immediately after initial contigs are formed, before repeat detection. Initial contigs that span a repeat can be split into multiple conitgs; none of these new contigs will be 'unassembled', even if they are a single read.

CHAPTER 6

Canu Command Reference

Every command, even the useless ones.

Commands marked as 'just usage' were automagically generated from the command line usage summary. Yes, some of them even crashed.

commands/bogart (just usage) The unitig construction algorithm. BOG stands for Best Overlap Graph; we haven't figured out what ART stands for.

commands/bogus (just usage) A unitig construction algorithm simulator. Given reads mapped to a reference, returns the largest unitigs possible.

commands/canu (just usage) The executive in charge! Coordinates all these commands to make an assembler.

commands/correctOverlaps (just usage) Part of Overlap Error Adjustment, recomputes overlaps given a set of read corrections.

commands/estimate-mer-threshold (just usage) Decides on a k-mer threshold for overlapInCore seeds.

commands/fastqAnalyze (just usage) Analyzes a FASTQ file and reports the best guess of the QV encoding. Can also rewrite the FASTQ to be in Sanger QV format.

commands/fastqSample (just usage) Extracts random reads from a single or mated FASTQ file. Extracts based on desired coverage, desired number of reads/pairs, desired fraction of the total, or desired total length.

commands/fastqSimulate (just usage) Creates reads with unbiased errors from a FASTA sequence.

commands/fastqSimulate-sort (just usage) Given input from fastqSimulate, sorts the reads by position in the reference.

commands/filterCorrectionOverlaps (just usage) Part of Read Correction, filters overlaps that shouldn't be used for correcting reads.

commands/findErrors (just usage) Part of Overlap Error Adjustment, generates a multialignment for each read, outputs a list of suspected errors in the read.

commands/gatekeeperCreate (just usage) Loads FASTA or FASTQ reads into the canu read database, gkpStore.

commands/gatekeeperDumpFASTQ (just usage) Outputs FASTQ reads fromt the canu read database, gkpStore.

commands/gatekeeperDumpMetaData (just usage) Outputs read and library metadata fromt the canu read database, gkpStore.

commands/gatekeeperPartition (just usage) Part of Consensus, rearranges the canu read database, gkpStore, to localize read to unitigs.

commands/generateCorrectionLayouts (just usage) Part of Read Correction, generates the multialignment layout used to correct reads.

commands/leaff (just usage) Not actually part of canu, but it came along with meryl. Provides random access to FASTA, FASTQ and gkpStore. Also does some analysis tasks. Handy Swiss Army knife type of tool.

commands/meryl (just usage) Counts k-mer occurrences in FASTA, FASTQ and gkpStore. Performs mathematical and logical operations on the resulting k-mer databases.

commands/mhapConvert (just usage) Convert mhap output to overlap output.

commands/ovStoreBucketizer (just usage) Part of the parallel overlap store building pipeline, loads raw overlaps from overlapper into the store.

commands/ovStoreBuild (just usage) Sequentially builds an overlap store from raw overlaps. Simplest to run, but slow on large datasets.

commands/ovStoreDump (just usage) Dumps overlaps from the overlap store, ovlStore.

commands/ovStoreIndexer (just usage) Part of the parallel overlap store building pipeline, finalizes the store, after sorting with ovStoreSorter.

commands/ovStoreSorter (just usage) Part of the parallel overlap store building pipeline, sorts overlaps loaded into the store by ovStoreBucketizer.

commands/overlapConvert (just usage) Reads raw overlapper output, writes overlaps as ASCII. The reverse of overlapImport.

commands/overlapImport (just usage) Reads ASCII overlaps in a few different formats, writes either 'raw overlapper output' or creates an ovlStore.

commands/overlapInCore (just usage) The classic overlapper algorithm.

commands/overlapInCorePartition (just usage) Generate partitioning to run overlapInCore jobs in parallel.

commands/overlapPair (just usage) An *experimental* algorithm to recompute overlaps and output the alignments.

commands/prefixEditDistance-matchLimitGenerate (just usage) Generate source code files with data representing the minimum length of a good overlap given some number of errors.

commands/splitReads (just usage) Part of Overlap Based Trimming, splits reads based on overlaps, specifically, looking for PacBio hairpin adapter signatures.

commands/tgStoreCoverageStat (just usage) Analyzes tigs in the tigStore, computes the classic arrival rate statistic.

commands/tgStoreDump (just usage) Analyzes and outputs tigs from the tigStore, in various formats (FASTQ, layouts, multialignments, etc).

commands/tgStoreFilter (just usage) Analyzes tigs in the tigStore, marks those that appear to be spurious 'degenerate' tigs.

commands/tgStoreLoad (just usage) Loads tigs into a tigStore.

commands/tgTigDisplay (just usage) Displays the tig contained in a binary multialignment file, as output by utgcns.

commands/trimReads (just usage) Part of Overlap Based Trimming, trims reads based on overlaps.

commands/utgcns (just usage) Generates a multialignment for a tig, based on the layout stored in tigStore. Outputs FASTQ, layouts and binary mutlialignment files.

CHAPTER 7

Software Background

Canu is derived from the Celera Assembler. The Celera assembler [Myers 2000] was designed to reconstruct mammalian chromosomal DNA sequences from the short fragments of a whole genome shotgun sequencing project. The Celera Assembler was used to produce reconstructions of several large genomes, namely those of Homo sapiens [Venter 2001], Mus musculus [Mural 2002], Rattus norvegicus [unpublished data], Canis familiaris [Kirkness 2003], Drosophila melanogaster [Adams 2000], and Anopheles gambiae [Holt 2001]. The Celera Assembler was shown to be very accurate when its reconstruction of the human genome was compared to independent reconstructions completed later [Istrail 2004]. It was used to reconstructing one of the first large-scale metagenomic projects [Venter 2004, Rusch 2007] and a diploid human reference [Levy 2007, Denisov 2008]. It was adapted to 454 Pyrosequencing [Miller 2008] and PacBio sequencing [Koren 2011], demonstrating finished bacterial genomes [Koren 2013] and efficient algorithms for eukaryotic assembly [Berlin 2015].

In 2015 Canu was forked from Celera Assembler and specialized for single-molecule high-noise sequences. The Celera Assembler codebase is no longer maintained.

Canu is a pipeline consisting of several executable programs and perl driver scripts. The source code includes programs in the C++ language with Unix make scripts. The original Celera Assembler was designed to run under Compaq(R) Tru64(R) Unix with access to 32GB RAM. It has also run under IBM(R) AIX(R) and Red Hat Linux.

The Celera Assembler was released under the GNU General Public License, version 2 as as supplement to the publication [Istrail 2004]. For the most recent license information please see README.licences

References

- Adams et al. (2000) The Genome Sequence of Drosophila melanogaster. Science 287 2185-2195.
- Myers et al. (2000) A Whole-Genome Assembly of Drosophila. Science 287 2196-2204.
- Venter et al. (2001) The Sequence of the Human Genome. Science 291 1304-1351.
- Mural et al. (2002) A Comparison of Whole-Genome Shotgun-Derived Mouse Chromosome 16 and the Human Genome. Science 296 1661-1671.
- Holt et al. (2002) The Genome Sequence of the Malaria Mosquito Anophelesd gambiae. Science 298 129-149.

- Istrail et al. (2004) Whole Genome Shotgun Assembly and Comparison of Human Genome Assemblies. PNAS 101 1916-1921.
- Kirkness et al. (2003) The Dog Genome: Survey Sequencing and Comparative Analysis. Science 301 1898-1903.
- Venter et al. (2004) Environmental genome shotgun sequencing of the Sargasso Sea. Science 304 66-74.
- Levy et al. (2007) The Diploid Genome Sequence of an Individual Human. PLoS Biology 0050254
- Rusch et al. (2007) The Sorcerer II Global Ocean Sampling Expedition: Northwest Atlantic through Eastern Tropical Pacific. PLoS Biology 1821060.
- Denisov et al. (2008) Consensus Generation and Variant Detection by Celera Assembler. Bioinformatics 24(8):1035-40
- Miller et al. (2008) Aggressive Assembly of Pyrosequencing Reads with Mates. Bioinformatics 24(24):2818-2824
- Koren et al. (2012) Hybrid error correction and de novo assembly of single-molecule sequencing reads, Nature Biotechnology, July 2012.
- Koren et al. (2013) Reducing assembly complexity of microbial genomes with single-molecule sequencing, Genome Biology 14:R101.
- Berlin et. al. (2015) Assembling Large Genomes with Single-Molecule Sequencing and Locality Sensitive Hashing. Nature Biotechnology. (2015).

Canu is a fork of the Celera Assembler designed for high-noise single-molecule sequencing (such as the PacBio RSII or Oxford Nanopore MinION).

CHAPTER 8

Publication

Koren S, Walenz BP, Berlin K, Miller JR, Phillippy AM. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome Research. (2017).

CHAPTER 9

Install

The easiest way to get started is to download a release. If you encounter any issues, please report them using the github issues page.

Alternatively, you can also build the latest unreleased from github:

```
git clone https://github.com/marbl/canu.git
cd canu/src
make -j <number of threads>
```

CHAPTER 10

Learn

- *Quick Start* - no experience or data required, download and assemble *Escherichia coli* today!
- *FAQ* - Frequently asked questions
- *Canu tutorial* - a gentle introduction to the complexities of canu.
- *Canu pipeline* - what, exactly, is canu doing, anyway?
- *Canu Parameter Reference* - all the paramters, grouped by function.
- *Canu Command Reference* - all the commands that canu runs for you.
- *Canu History* - the history of the Canu project.