FALCON

Release 0.5

Greg Concepcion, Sarah Kingan, Chris Dunn, Jason Chin

Sep 10, 2018

Contents

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 1

About FALCON

1.1 Overview

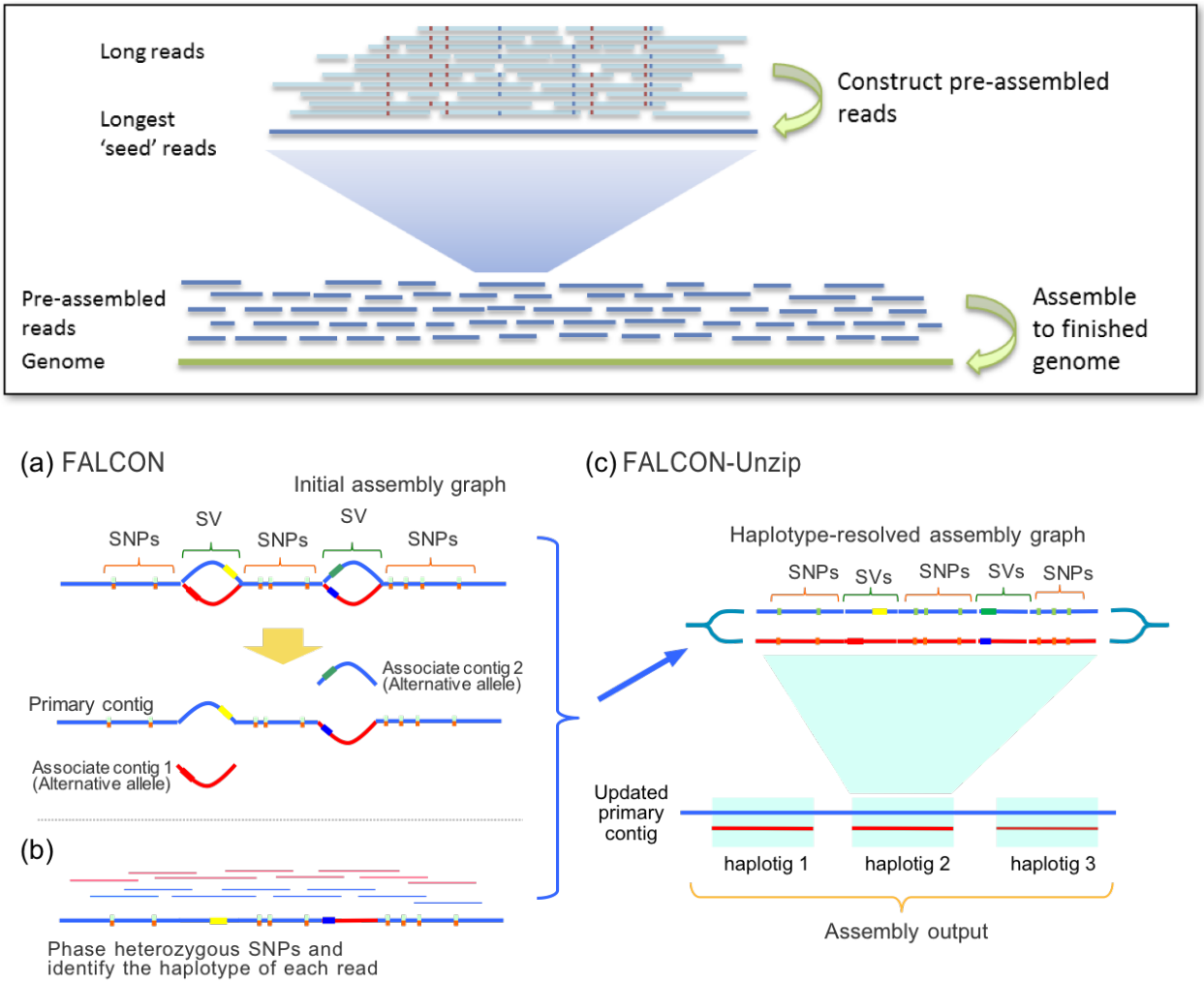
FALCON and FALCON-Unzip are *de novo* genome assemblers for PacBio long reads, also known as single-molecule real-time (SMRT) sequences. FALCON is a diploid-aware assembler which follows the hierarchical genome assembly process (HGAP) and is optimized for large genome assembly (e.g. non-microbial). FALCON produces a set of primary contigs (a-contigs), which represent divergent allelic variants. Each a-contig is associated with a homologous genomic region on an p-contig.

FALCON-Unzip is a true diploid assembler. It takes the contigs from FALCON and phases the reads based on heterozygous SNPs identified in the initial assembly. It then produces a set of partially phased *primary contigs* and fully phased *haplotigs* which represent divergent haplotyes.

1.2 Detailed Description

The hierarchical genome assembly process proceeds in two rounds. The first round of assembly involves the selection of seed reads, or the longest reads in the dataset (user-defined *length_cutoff*). All shorter reads are aligned to the seed reads, in order to generate consensus sequences with high accuracy. We refer to these as pre-assembled reads but they can also be thought of as "error corrected" reads. During the pre-assembly process, seed reads may be split or trimmed at regions of low read coverage (user-defined *min_cov* for *falcon_sense_option*). The performance of the pre-assembly process is captured in the pre-assembly stats file.

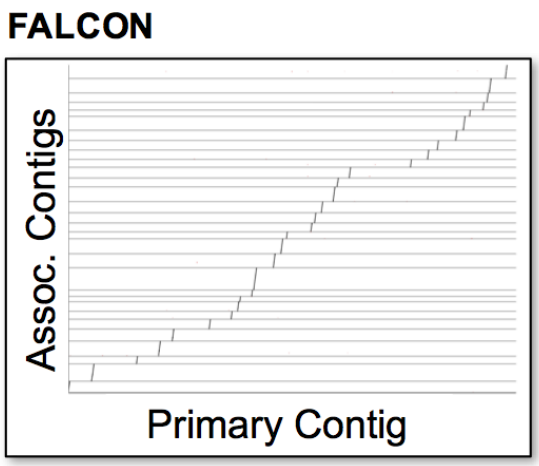
In the next round of HGAP, the *preads*, are aligned to each other and assembled into genomic contigs.

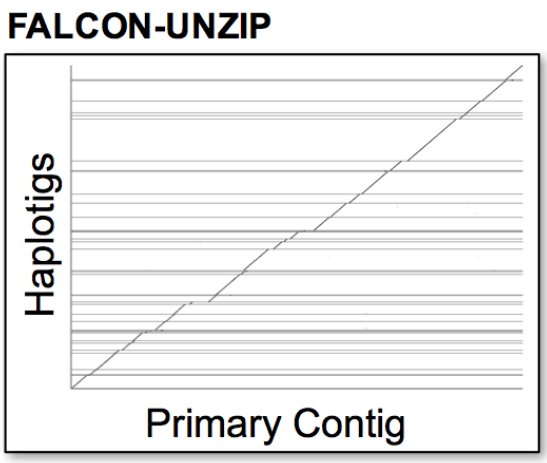


For more complex genomes assembled with `FALCON`, "bubbles" in the contig-assembly graph that result from structural variation between haplotypes may be resolved as associate and primary contigs. The unzip process will extend haplotype phasing beyond "bubble" regions, increasing the amount of phased contig sequence. It is important to note that while individual haplotype blocks are phased, phasing does not extend **between** haplotigs. Thus, in part **C**) of the figure above, **haplotig_1** and **haplotig_2** may originate from different parental haplotypes. Additional information is needed to phase the haplotype blocks with each other.

Associate contig IDs contain the name of their primary contig but the precise location of alignment must be determined with third party tools such as NUCmer. For example, in a `FALCON` assembly, *000123F-010-01* is an associated contig to primary contig *000123F*. In a `FALCON-Unzip` assembly, *000123F_001* is a haplotig of primary contig *000123F*.

Below are examples of alignments between associate and primary contigs from `FALCON`, and haplotigs and primary contigs from `FALCON-Unzip`. Alignments were built with NUCmer and visualized with Assemblytics. Precise coordinates may be obtained with the show-coords utilty from MUMmer.





1.3 Choosing an Assembler: HGAP4 vs FALCON vs FALCON-Unzip

1.3.1 HGAP4

We recommend `HGAP4`, part of the SMRT Link web-based analysis suite, for genomes of known complexity, no larger than human (3Gb or smaller), although underlying compute resources for your SMRT Link instance will influence performance and feasibility. The assembly process for `HGAP4` in the SMRT Link GUI (graphical user interface) is identical to `FALCON` at the command line, besides differences in compute resource configuration and minor differences in directory structure. The `HGAP4` pipeline by default includes a round of genome "polishing" which employs the `resequencing` pipeline.

`HGAP4` RESULTS ARE NOT COMPATIBLE WITH `FALCON-Unzip` AT THIS TIME!

`HGAP4` inputs are a PacBio subread BAM dataset, either Sequel or RSII. The FASTA and FASTQ files output from `HGAP4` are a concatenation of the primary and associate contigs, which are output from `FALCON` as separate files.

1.3.2 Command Line

Users more comfortable at the command line may use `FALCON` for genomes of any size or complexity. Command line inputs are FASTA files of Sequel or RSII subreads. Command-line `FALCON` does not automatically polish the assembly. If a user wishes, assembly polishing may be run using the `resequencing` pipeline of pbsmrtpipe (available for command-line installation using the SMRT_Link download, see SMRT_Tools_Reference_Guide for installation instructions). Resequencing requires PacBio subread BAM inputs.

We recommend the `FALCON-Unzip` module for heterozygous or outbred organisms that are diploid or higher ploidy. Users wishing to run `FALCON-Unzip` must do so only after running `FALCON` on the command line. `HGAP4` IS NOT COMPATIBLE WITH `FALCON-UNZIP`! The `FALCON-Unzip` module requires both FASTA and PacBio BAM inputs for subreads.

1.4 References

Chin et al. (2016). Phased diploid genome assembly with single-molecule real-time sequencing. Nature Methods. 13(12), 1050.

Chin, et al. (2013). Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. Nature Methods. 10(6), 563.

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 2

Quick Start Guide

2.1 Installation

The quickest way to install FALCON + FALCON_unzip is to download and run this install script:

Install script	Tarball date	Status
`install_unzip.sh`	*3/12/2018*	
`install_unzip.sh`	*8/08/2018*	Beta

```
$ bash -ex install_unzip.sh /path/to/your/install/dir
```

Note: This will clone the FALCON-integrate repository, FALCON_unzip binaries, build a virtualenv and launch a small test case assembly to ensure successful installation.

samtools and minimap2 must be installed separately and in your $PATH.

If you don't see any errors, you will have installed FALCON/FALCON_unzip and successfully assembled and unzipped a small test dataset. At this point you should be ready to confidently launch a larger genome assembly.

To activate your FALCON_unzip virtualenv in the future:

```
$ source /path/to/your/install/dir/fc_env/bin/activate
```

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly


falcon

CHAPTER 3

Pipeline

3.1 FALCON

A FALCON job can be broken down into 3 steps:

1. *Overlap detection and error correction of rawreads*
2. *Overlap detection between corrected reads*
3. *String Graph assembly of corrected reads*

Each step is performed in it's own subdirectory within the FALCON job

```
falcon_job/
    ├── 0-rawreads/      # Raw read error correction directory
    ├── 1-preads_ovl/    # Corrected read overlap detection
    ├── 2-asm-falcon/    # String Graph Assembly
    ├── mypwatcher/      # Job scheduler logs
    ├── scripts/
    └── sge_log/         # deprecated
```

The assembly process is driven by the script fc_run.py which should be sent to the scheduler or run on a head node as it needs to persist throughout the entire assembly process. It takes as input a single *config* file typically named *fc_run.cfg*, which references a list of fasta input files. The config file can be configured to run locally, or submit to a job scheduler. However, if your dataset is anything larger than a bacterial sized genome and unless you've tuned your system specifically for the organism you're trying to assemble, then most likely you should be running on a cluster in order to more effectively leverage your computational resources.

The configuration file also allows you to control other aspects of your job such as how your compute resources are distributed as well as set many parameters to help you reach an "optimized" assembly according to the nature of your input data. Unfortunately at this point there is no "magic" way to auto-tune the parameters so you should probably spend some time in the *Configuration* section to understand what options are available to you. Some example configuration files can be found *here*

3.1.1 Step 1: Overlap detection and error correction of raw reads

The first step of the pipeline is to identify all overlaps in the raw reads. Currently this is performed with a modified version of Gene Myers' DALIGNER.

In order to identify overlaps, your raw reads must first be converted from fasta format into a dazzler database. This is a very I/O intensive process and will be run from the node where `fc_run.py` was executed. If this is an issue, you should submit the command with a wrapper script to your grid directly.

Once the database has been created and partitioned according to the parameters set in your *fc_run.cfg*, an all vs all comparison of the reads must be performed. Accordingly, due to the all vs all nature of the search this is the most time consuming step in the assembly process. To walk through the actual steps of this part of the pipeline you should take a look at `0-rawreads/prepare_rdb.sub.sh`. Essentially it consists of running:

1. *fasta2DB* to format the database
2. *DBsplit* to partition the database
3. *HPC.daligner* to generate the daligner commands necessary for all-vs-all comparison

After overlaps have been detected, you will be left with many `job_*` directories full of alignment files `*.las` containing the information about the overlaps. After merging the alignment files (see `m_*` directories), the next step is to error correct the reads leveraging the overlap information. In the `0-rawreads/preads` directory you will find a series of scripts for performing the error correction. The process basically consists of using `LA4Falcon` with a length cutoff and piping the output to fc_consensus.py to generate a fasta file with corrected reads.

```
0-rawreads/
    ├── job_*                      # dirs for all of the daligner jobs
    ├── m_*/                       # dirs for all of the LA4Merge jobs
    ├── preads/                    # sub-dir for preads generation
    ├── report/                          # pre-assembly stats
    ├── cns-scatter/                  # dir of scripts for falcon-consensus jobs
    ├── daligner-scatter/          # dir of scripts for daligner jobs
    ├── merge-scatter/                # dir of scripts for LAMerge jobs
    ├── merge-gather/                 # dir of scripts for gathering LAMerge inputs
    ├── raw-gather/                # dir of scripts for gathering daligner jobs for␣
↪merging
    ├── input.fofn                # list if your input *.fasta files
    ├── length_cutoff              # text file with length cutoff for seed reads
    ├── pwatcher.dir                  # dir of individual pipeline jobs stderr and␣
↪stdout
    ├── prepare_rdb.sh             # env wrapper script
    ├── raw_reads.db               # dazzler DB file
    ├── raw-fofn-abs                  # dir of scripts for gathering raw reads inputs
    ├── rdb_build_done             # database construction sentinel file
    ├── run_jobs.sh               # listing of all overlap step commands
    ├── run.sh                           # masker job script
    ├── run.sh.done                   # sentinel file for all jobs
    ├── task.json                    # json file specifying inputs, outputs, and␣
↪params
    └── task.sh                             # script to run json file
```

The following parameters affect this step directly:

- *sge_option_da*
- *sge_option_la*
- *pa_concurrent_jobs*
- *cns_concurrent_jobs*

- *pa_DBsplit_option*
- *falcon_sense_option*

3.1.2 Step 2: Overlap detection of corrected reads

The only conceptual difference between the first and second overlap steps is that consensus calling is not performed in the second step. After *pread* overlap detection, it's simply a matter of extracting the information from the corrected reads database with `DB2Falcon -U preads`.

Depending on how well the error-correction step proceeded as well as the how much initial coverage was fed into the pipeline (e.g. *length_cutoff*), the input data for this step should be significantly reduced and thus, the second overlap detection step will proceed significantly faster.

The commands in this step of the pipeline are very similar to before albeit with different parameter settings to account for the reduced error-rate of the *preads*. See the driver script `prepare_pdb.sub.sh` for details on actual parameter settings used.

```
1-preads_ovl/
    ├── job_*/                  # directories for daligner jobs
    ├── m_*/                    # directories for LA4Merge jobs
    ├── db2falcon/              # dir of scripts for formatting preads for falcon
    ├── gathered-las/           # dir of scripts for gathering daligner jobs
    ├── merge-gather/           # dir of scripts for gathering LAMerge inputs
    ├── merge-scatter/          # dir of scripts for LAMerge jobs
    ├── daligner-scatter/       # dir of scripts for daligner jobs
    ├── pdb_build_done          # sentinel file for pread DB building
    ├── preads.db               # preads dazzler DB
    ├── prepare_pdb.sh          # env wrapper script
    ├── pwatcher.dir            # dir of individual pipeline jobs stderr and stdout
    ├── run_jobs.sh             # listing of all pread overlap job commands
    ├── run.sh                  # masker job script
    ├── run.sh.done             # sentinel file for all jobs
    ├── task.json               # json file specifying inputs, outputs, and params
    └── task.sh                 # script to run json file
```

The following parameters affect this step directly:

- *sge_option_pda*
- *sge_option_pla*
- *ovlp_concurrent_jobs*
- *ovlp_DBsplit_option*
- *ovlp_HPCdaligner_option*

3.1.3 Step 3: String Graph assembly

The final step of the FALCON Assembly pipeline is generation of the final *String Graph* assembly and output of contig sequences in fasta format. Four commands are run in the final phase of FALCON:

1. *fc_ovlp_filter* - Filters overlaps based on the criteria provided in *fc_run.cfg*
2. *fc_ovlp_to_graph* - Constructs an overlap graph of reads larger than the length cutoff
3. *fc_graph_to_contig* - Generates fasta files for contigs from the overlap graph.
4. *fc_dedup_a_tigs* - Removes duplicate associated contigs

You can see the details on the parameters used by inspecting `2-asm_falcon/run_falcon_asm.sub.sh` This step of the pipeline is very fast relative to the overlap detection steps. Sometimes it may be useful to run several iterations of this step with different parameter settings in order to identify a "best" assembly.

The final output of this step is a fasta file of all of the primary contigs, `p_ctg.fa` as well as an associated contig fasta file, `a_ctg.fa` that consists of all of the structural variants from the primary contig assembly.

```
2-asm-falcon/
    ├── a_ctg_all.fa                # all associated contigs, including duplicates
    ├── a_ctg_base.fa               #
    ├── a_ctg_base_tiling_path      #
    ├── a_ctg.fa                    # De-duplicated associated fasta file
    ├── a_ctg_tiling_path           # tiling path informaiton for each associated␣
↪contig
    ├── falcon_asm_done             # FALCON Assembly sentinal file
    ├── p_ctg.fa                    # Fasta file of all primary contigs
    ├── p_ctg_tiling_path           # Tiling path of preads through each primary␣
↪contig
    ├── c_path                      #
    ├── ctg_paths                   # corrected read paths for each contig
    ├── fc_ovlp_to_graph.log        # logfile for process of converting overlaps to␣
↪assembly graph
    ├── utg_data                    #
    ├── sg_edges_list               # list of all edges
    ├── chimers_nodes               #
    ├── preads.ovl                  # List of all overlaps between preads
    ├── run_falcon_asm.sh           # env wrapper script
    ├── task.json                       # json file specifying inputs, outputs, and␣
↪params
    ├── task.sh                             # script to run json file
    ├── run.sh.done                     # sentinel file for all jobs
    └── run.sh                      # Assembly driver script
```

The following parameters affect this step directly:

- *sge_option_fc*
- *overlap_filtering_setting*
- *length_cutoff_pr*

3.2 FALCON_unzip

FALCON_unzip operates from a completed FALCON job directory. After tracking the raw reads to contig, A FALCON_unzip job can be broken down into 3 steps

1. *Identify SNPs and assign phases*
2. *Annotate Assembly graph with Phases*
3. *Graph building*

```
3-unzip/
├── 0-phasing/                  # Contig phasing jobs
├── 1-hasm/                     # Contig Graph assembly information
├── read_maps/                  # rawread_to_contigs; read_to_contig_map
├── reads/                      # raw read fastas for each contig
├── all_p_ctg.fa                # partially phased primary contigs
```

(continues on next page)

(continued from previous page)

```
├── all_h_ctg.fa                # phased haplotigs
├── all_p_ctg_edges             # primary contig edge list
├── all_h_ctg_edges             # haplotig edge list
├── all_h_ctg_ids               # haplotig id index
└── all_phased_reads            # table of all phased raw reads
```

3.2.1 Step 1: Identify SNPs and assign phases

Inside of `0-phasing/` you vill find a number of directories for each contig. Each contains the scripts to map the raw reads to the contigs and subsequently identify SNPs. The generated SNP tables can subsequently be used to assign phases to reads.

3.2.2 Step 2: Graph annotation and haplotig

Inside of `1-hasm/` you can find the driver script `hasm.sh` which contains the commands necessary to filter overlaps and traverse the assembly graph paths and subsequently output phased contig sequence. Assembly Graphs for each contig as well as fasta files for the partially phased primary contigs and fully phased haplotigs can be found in each `1-hasm/XXXXXXF` directory.

3.2.3 Step 3: Call Consensus (Optional)

Finally, the `FALCON_unzip` pipeline can optionally be used to run quiver and call high quality consensus. This step takes as input the primary contig and haplotig sequences output in the previous step. For convenience, these files have all been concatenated together into `3-unzip/all_p_ctg.fa` and `3-unzip/all_h_ctg.fa` respectively. The final consensus output can be found in `falcon_jobdir/4-quiver/cns_output/*.fast[a|q]`. In order to run the consensus step as part of the FALCON_unzip pipeline, You need to provide the input_bam_fofn *fc_unzip.cfg* option in order for this to work.

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 4

Commands

4.1 FALCON Commands

DB2Falcon Used to dump dazzler preads.db into FASTA format for subsequent *String Graph* assembly

fc_run.py This script drives the entire assembly process

fc_consensus.py `fc_consensus` has many options. You can use the parameter falcon_sense_option to control it. In most cases, the `--min_cov` and `--max_n_read` are the most important options. `--min_cov` controls when a seed read gets trimmed or broken due to low coverage. `--max_n_read` puts a cap on the number of reads used for error correction. In highly repetitive genome, you will need to make the value for `--max_n_read` smaller to make sure the consensus code does not waste time aligning repeats. The longest proper overlaps are used for correction to reduce the probability of collapsed repeats.

fc_dedup_a_tigs.py remove duplicated *associated contigs*, mostly induced by tandem repeat alignment uncertainty

fc_graph_to_contig.py Generate contigs based on assembly graph

fc_ovlp_to_graph.py Generate an assembly graph given a list of overlapping preads.

fc_ovlp_filter.py Filter overlaps based on given criteria

4.2 FALCON_unzip commands

fc_get_read_hctg_map.py Generate a read-to-contig map

fc_dedup_h_tigs.py

fc_graphs_to_h_tigs.py

fc_ovlp_filter_with_phase.py

fc_phased_ovlp_to_graph.py

fc_phasing.py

fc_phasing_readmap.py

fc_quiver.py

fc_rr_hctg_track.py

fc_select_reads_from_bam.py

fc_track_reads_htigs.py

fc_unzip.py

4.3 Dazzler commands

These commands are part of Gene Meyer's Dazzler Suite of tools Dazzler Blog

FALCON relies on a slightly modified version of Gene Meyer's code that can be found here, but is also bundled with the FALCON-integrate github repository.

daligner: Compare subject sequences to target sequences `daligner` is controlled by *pa_HPCdaligner_option* and *ovlp_HPCdaligner_option*.

To limit memory, one can use the `-M` option. For human assembly, we've tested with `-M 32` for using 32G RAM for each daligner. Other possibilities are under investigation.

For more details on daligner options, see the Dazzler Blog

DB2fasta: The set of .fasta files for the given DB are recreated from the DB exactly as they were input.

DBdump: Like DBshow, DBdump allows one to display a subset of the reads in the DB and select which information to show about them including any mask tracks.

DBdust: Runs the symmetric DUST algorithm over the reads in the untrimmed DB

DBsplit: The total number of jobs that are run is determined by how one "splits" the sequence database. You should read Gene Myers's blog *Dazzler Blog <http://dazzlerblog.wordpress.com>* carefully to understand how the tuning options, *pa_DBsplit_option* and *pa_HPCdaligner_option* work. Generally, for large genomes, you should use `-s400` (400Mb sequence per block) in *pa_DBsplit_option*. This will make a smaller number of jobs but each job will run longer. However, if you have a job scheduler which limits how long a job can run, it might be desirable to have a smaller number for the `-s` option.

DBstats: Show overview statistics for all the reads in the trimmed data base <path>.db

fasta2DB: Convert a fasta to a dazzler DB.

HPC.daligner: Generates overlap script to run all necessary daligner, LAsort and LAmerge commands

LA4Falcon: Output data from a Dazzler DB into fasta format for FALCON. You can supply the argument `-H` with an integer value to filter reads below a given threshold.

LAcheck: Check integrity of alignment files.

LAmerge: Merge the .las files <parts> into a singled sorted file

LAsort: Sort alignment files

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 5

Tutorial

In this section we will run the FALCON pipeline on an *E. coli* dataset. We will work through the commands and results and give you ideas of how to assess the perfomance of FALCON on your dataset so you can modify parameters and trouble-shoot more effectively. This tutorial is a beginners guide to FALCON but assumes bioinformatics fluency.

5.1 Input Files

You will need three types of files to get started, your PacBio data in fasta format (can be one or many files), a text file telling FALCON where to find your fasta files, and a *configuration* file. All files except the fasta files must be in your job directory.

5.1.1 1. Download *E. coli* dataset

An example *E. coli* dataset can be download from here. and then unpacked. e.g.:

```
$ wget https://downloads.pacbcloud.com/public/data/git-sym/ecoli.m140913_050931_42139_
↪c100713652400000001823152404301535_s1_p0.subreads.tar.gz
$ tar -xvzf ecoli.m140913_050931_42139_c100713652400000001823152404301535_s1_p0.
↪subreads.tar.gz
```

You should find three fasta files of ~350 Mb each in the newly created subdirectory.

5.1.2 2. Create FOFN

Next, create a "file-of-file-names", ("fofn") with the full path of each fasta file, one per line.

```
/my/path/to/data/ecoli.1.subreads.fasta
/my/path/to/data/ecoli.2.subreads.fasta
/my/path/to/data/ecoli.3.subreads.fasta
```

5.1.3 3. Download configuration file

If you are running on a cluster with a scheduler use this a starting point: `fc_run_ecoli.cfg` If you are running your job locally, try this file: `fc_run_ecoli_local.cfg`

These config files are meant to be starting points only! You will need to make adjustments according to your particular compute setup. Please refer to these *config* files as you plan your run.

Note that I have manually specified a seed read length cutoff of 15kb rather than using an automated cut off (*length_cutoff* = -1, with calculated length cut off = 22486). The raw read coverage is very high (>200X); by reducing the seed read length cutoff, we avoid enriching our seed reads for erroneous chimeric (and very long) reads. Try running the assembly using the automated seed read length cut off, you should get a fragmented (28 contigs) and incomplete assembly (< 900Mb).

5.2 Running FALCON

I send all of my `FALCON` jobs to the scheduler for ease of tracking job progress. Here is an example bash script `run_falcon.sh` that submits to an SGE cluster:

```
#!/bin/bash
#$ -S /bin/bash
#$ -N myJob
#$ -cwd
#$ -q myqueue

# load dependencies
module load python/2.7.9 gcc/4.9.2

# source build
cd /path_to_build/src/FALCON-integrate/
source env.sh

# navigate to job directory, directory containing input.fofn
cd /path/to/my/job_dir

# run it!
fc_run fc_run.cfg
```

To initiate the `FALCON` run, I just submit my job to the scheduler with a qsub command:

```
$ qsub run_falcon.sh
```

Alternatively, you can add the `fc_env/bin` directory to your `$PATH` and invoke `fc_run` at the command line with your `fc_run.cfg` as the argument. Note that this shell needs to persist through the entire assembly process so you may need to use a window manager like screen to maintain your connection.

```
falcon_jobdir$ export PYTHONUSERBASE=/path_to_build/fc_env/
falcon_jobdir$ export PATH=$PYTHONUSERBASE/bin:$PATH
falcon_jobdir$ fc_run fc_run.cfg
```

5.3 Assessing Run Progress

Refer to the pipeline document for detailed summary of `FALCON` job directory structure, sequence of commands, and output files created.

5.3.1 Counting Completed Jobs

The majority of run-time is spent during the daligner phases, performing the alignments and then sorting and merging them. To determine how many jobs are performed for each step, refer to `0-rawreads/run_jobs.sh`.

```
$ grep '^#' 0-rawreads/run_jobs.sh

    # Daligner jobs (60)
    # Initial sort jobs (400)
    # Check initial .las files jobs (80) (optional but recommended)
    # Remove initial .las files
    # Level 1 merge jobs (20)
    # Check level 2 .las files jobs (20) (optional but recommended)
    # Remove level 1 .las files (optional)
```

To determine how many jobs have completed, count the sentinel files that indicate a job is complete. For example:

```
$ find 0-rawreads/ -name "job*done" | wc -l
60

$ find 0-rawreads/ -name "m_*done" | wc -l
20
```

5.4 Assessing Run Performance

5.4.1 Raw and Pread Coverage and Quality

The *E. coli* subreads are a total of 1.01 Gb of data in 105,451 reads. `countFasta.pl` is a useful script by Joseph Fass and Brad Sickler at UC Davis for calculating total sequence length and other assembly metrics).

You can confirm that your dazzler database was correctly constructed using a utility from the dazzler suite:

```
$ DBstats raw_reads.db > raw_reads.stats
$ head raw_reads.stats -n 17

Statistics for all reads of length 500 bases or more

        90,747 reads        out of         105,451  ( 86.1%)
        964,281,429 base pairs   out of   1,013,118,375  ( 95.2%)

        10,626 average read length
        6,805 standard deviation

Base composition: 0.248(A) 0.242(C) 0.263(G) 0.246(T)

Distribution of Read Lengths (Bin size = 1,000)

        Bin:      Count  % Reads  % Bases     Average
        45,000:     1      0.0      0.0       45611
```

You can see that we discarded 13.9% of the raw bases and 4.8% of the reads by employing a raw read length cut off of 500bp in the *DBsplit* options. This file can also be used to plot a histogram of raw read lengths.

The genome of this *E. coli* strain is 4.65 Mb long for a raw read coverage of ~207 fold. Confirm this with the preassembly report:

```
$ cat 0-rawreads/report/pre_assembly_stats.json

"genome_length": 4652500,
"length_cutoff": 15000,
"preassembled_bases": 350302363,
"preassembled_coverage": 75.293,
"preassembled_mean": 10730.33,
"preassembled_n50": 16120,
"preassembled_p95": 22741,
"preassembled_reads": 32646,
"preassembled_seed_fragmentation": 1.451,        # number split preads / seed reads
"preassembled_seed_truncation": 4453.782,        # ave bp lost per pread due to low cov
"preassembled_yield": 0.758,                     # total pread bp / seed read bp
"raw_bases": 964281429,
"raw_coverage": 207.261,
"raw_mean": 10626.042,
"raw_n50": 14591,
"raw_p95": 23194,
"raw_reads": 90747,
"seed_bases": 461851093,
"seed_coverage": 99.269,                         # raw base coverage depth on seed␣
↪reads
"seed_mean": 20029.103,
"seed_n50": 19855,
"seed_p95": 28307,
"seed_reads": 23059
```

A note on these statistics: in the process of created preads, seeds reads with insufficient raw read coverage (usually due to base errors) will be split or truncated. The preassembled seed fragmentation, truncation, and yield stats summarize the quality of pread assembly. A good preassembled yield should be greater than 50%. Note that if an automated seed read length is used for this data (22486), preassembled seed read truncation is ~6kb, indicating that many of the longest raw reads are not supported by the rest of the data.

You can similarly summarize the contents of the dazzler database for preads using DBstats and plotting in R.

5.4.2 Contig Stats

When your run is complete, you can summarize your assembly stats using the `countFasta.pl` script:

```
$ countFasta.pl p_ctg.fa > p_ctg.stats
$ countFasta.pl a_ctg.fa > a_ctg.stats
$ tail p_ctg.stats

Total length of sequence:       4635395 bp
Total number of sequences:      1
N25 stats:                      25% of total sequence length is contained in␣
↪the 1 sequences >= 4635395 bp
N50 stats:                      50% of total sequence length is contained in␣
↪the 1 sequences >= 4635395 bp
N75 stats:                      75% of total sequence length is contained in␣
↪the 1 sequences >= 4635395 bp
Total GC count:                 2352187 bp
GC %:                           50.74 %
```

5.4.3 Assembly Graph and Pread Overlaps

Assembly contiguity can be enhanced by adjusting a few parameters in the last stage of the assembly process. You can try a grid of *pread length cut offs* for the filtering of the final overlaps in the assembly graph. In a general sense, longer pread length cut offs will increase the contiguity (contig N50) in your assembly, but may result in shorter over all assembly length. To try different length cut off, rename your 2-asm-falcon dir, modify the config file, rename the log and mypwatcher directory, and restart `FALCON`:

```
$ mv 2-asm-falcon 2-asm-falcon_12kb
$ mv mypwatcher/ mypwatcher0/
$ mv all.log all0.log
$ qsub run_falcon.sh
```

The other parameter to adjust is the number of overlaps in the assembly graph. First, look at a histogram of the number of overlaps on the 5' and 3' end of each read. Run the falcon utility:

```
# make sure utility is in $PATH
$ export PYTHONUSERBASE=/path_to_build/fc_env/
$ export PATH=$PYTHONUSERBASE/bin:$PATH

# navigate to directory
$ cd 2-asm-falcon
$ fc_ovlp_stats --fofn ../1-preads_ovl/merge-gather/las.fofn > ovlp.stats
```

Then plot histograms of the number of 5' and 3' overlaps between preads in R. This can inform your parameters for *sge_option_fc* where `min_cov` and `max_cov` should flank the bulk of the distribution. For repetative genomes, a second mode in the distribution may appear, containing preads ending or begining in repetative material. It is best to choose a `max_cov` to the left of the repeat mode that removes these repetative overlaps.

5.5 Troubleshooting Run

If you find your run has died here are some suggestions of how to restart, in order of increasing difficulty:

5.5.1 Simple Restart

Simply rename your log file and `mypwatcher` directory and restart the pipeline. Renaming these files preserves them for you reference, and by removing the original mypwatcher directory the pipeline, when restarted, will scan your job directory for completed jobs and pick up where it left off:

```
$ mv mypwatcher/ mypwatcher0/
$ mv all.log all0.log
$ qsub run_falcon.sh
```

5.5.2 Directory Cleanup and Restart

First, determine which job caused the run to fail. For example:

```
$ grep 'ERROR' all.log

2016-11-21 03:21:39,482 - pypeflow.simple_pwatcher_bridge - ERROR - Task Node(0-
↪rawreads/m_00210) failed with exit-code=99
2016-11-21 03:21:39,482 - pypeflow.simple_pwatcher_bridge - ERROR - Failed to clean-
↪up FakeThread: jobid=Pcfbdb8b3c50d5e status='EXIT '
```

(continues on next page)

(continued from previous page)

Delete all directories that failed, then rename the log file and `mypwatcher` as above:

```
$ rm -rf 0-rawreads/m_00210
$ mv mypwatcher/ mypwatcher0/
$ mv all.log all0.log
$ qsub run_falcon.sh
```

You can find out more details about the failed jobs in `mypwatcher/` to diagnose the problem.

```
$ less mypwatcher/jobs/Pcfbdb8b3c50d5e/stderr
$ less mypwatcher/jobs/Pcfbdb8b3c50d5e/stdout
```

5.5.3 Manual Running of Failed Jobs

If your job still fails, try manually running the problematic jobs. Search in the job directory for the shell script containing the individual tasks and try manually running the shell script or individual tasks:

```
$ ls job_0000

job_0000_done  L1.19.5.las  L1.19.7.las  L1.5.19.las  L1.7.19.las   raw_reads.db  run.
↪sh       task.json
L1.19.4.las    L1.19.6.las  L1.4.19.las  L1.6.19.las  pwatcher.dir  rj_0000.sh    run.
↪sh.done  task.sh

$ head job_0000/rj_0000.sh -n 12

#!/bin/bash
set -vex

db_dir=/lustre/hpcprod/skingan/FALCON_tutorial/ecoli/0-rawreads
ln -sf ${db_dir}/.raw_reads.bps .
ln -sf ${db_dir}/.raw_reads.idx .
ln -sf ${db_dir}/raw_reads.db .
ln -sf ${db_dir}/.raw_reads.dust.anno .
ln -sf ${db_dir}/.raw_reads.dust.data .
daligner -v -t16 -H22486 -e0.7 -s1000 raw_reads.19 raw_reads.4 raw_reads.5 raw_reads.
↪6 raw_reads.7
LAcheck -v raw_reads *.las
LAsort -v raw_reads.4.raw_reads.19.C0 raw_reads.4.raw_reads.19.N0 raw_reads.4.raw_
↪reads.19.C1 raw_reads.4.raw_reads.19.N1 raw_reads.4.raw_reads.19.C2 raw_reads.4.raw_
↪reads.19.N2 raw_reads.4.raw_reads.19.C3 raw_reads.4.raw_reads.19.N3 && LAmerge -v␣
↪L1.4.19 raw_reads.4.raw_reads.19.C0.S raw_reads.4.raw_reads.19.N0.S raw_reads.4.raw_
↪reads.19.C1.S raw_reads.4.raw_reads.19.N1.S raw_reads.4.raw_reads.19.C2.S raw_reads.
↪4.raw_reads.19.N2.S raw_reads.4.raw_reads.19.C3.S raw_reads.4.raw_reads.19.N3.S
```

Once these jobs have run to completion, you can try restarting the pipeline.

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 6

Parameters

6.1 Configuration

Here are some example `fc_run.cfg` and `fc_unzip.cfg` files. We make no guarantee that they will work with your dataset and cluster configuration. We merely provide them as starting points that have proven themselves on internal datasets. A lot of your success will depend purely on the quality of the input data prior to even engaging the FALCON pipeline. Also, these particular configs were designed to work in our SGE compute cluster, so some tuning will likely be necessary on your part. You should consult with your HPC administrator to assist in tuning to your cluster.

6.1.1 FALCON Parameter sets

`fc_run_fungal.cfg` - Has worked well on a 40Mb fungal genome

`fc_run_human.cfg` - Has worked well on at least one human dataset

`fc_run_bird.cfg` - Has worked well on at least one avian dataset

`fc_run_yeast.cfg` - Has worked well on at least one yeast dataset

`fc_run_dipteran.cfg` - Has worked well on at least one dipteran (insect) dataset

`fc_run_mammal.cfg` - Has worked well on at least one mammalian dataset

`fc_run_mammalSequel.cfg` - Has worked well on at least one mammalian Sequel dataset

`fc_run_plant.cfg` - Has worked well on at least one plant (Ranunculales) dataset

`fc_run_arabidopsis.cfg` - Configuration for arabidopsis assembly in Chin et al. 2016

`fc_run_ecoli.cfg` - Configuration for test *E. coli* dataset

`fc_run_ecoli_local.cfg` - Configuration for test *E. coli* dataset run locally

6.1.2 FALCON_unzip Parameter sets

`fc_unzip.cfg` - General all purpose unzip config

6.2 Available Parameters

6.2.1 fc_run.cfg

input_fofn <str> filename for the file-of-filenames (fofn) Each line is fasta filename. Any relative paths are relative to the location of the input_fofn.

input_type <str> "raw" or "preads"

genome_size <int> estimated number of base-pairs in haplotype

seed-coverage <int> requested coverage for auto-calculated cutoff

length_cutoff <int> Raw reads shorter than this cutoff won't be considered in the assembly process. If '-1', then auto-calculate the cutoff based on genome_size and seed_coverage.

length_cutoff_pr <int> minimum length of seed-reads used after pre-assembly, for the "overlap" stage

target <str> "assembly" or "preads" If "preads", then pre-assembly stage is skipped and input is assumed to be preads.

default_concurrent_jobs <int> maximum concurrency This applies even to "local" (non-distributed) jobs.

pa_concurrent_jobs <str> Concurrency settings for pre-assembly

cns_concurrent_jobs <str> Concurrency settings for consensus calling

One can use cns_concurrent_jobs to control the maximum number of concurrent consensus jobs submitted to the job management system. The `out.XXXXX.fasta` files produced are used as input for the next step in the pipeline.

ovlp_concurrent_jobs <str> Concurrency settings for Overlap detection

job_type <str> grid submission system, or "local" Supported types include: "sge", "lsf", "pbs", "torque", "slurm", "local" case-insensitive

job_queue <str> grid job-queue name Can be overridden with section-specific sge_option_*

sge_option_da <str> Grid concurrency settings for initial daligner steps `0-rawreads/`

sge_option_la <str> Grid concurrency settings for initial las-merging `0-rawreads/`

sge_option_cns <str> Grid concurrency settings for error correction consensus calling

sge_option_pda <str> Grid concurrency settings for daligner on preads `1-preads_ovl/`

sge_option_pla <str> Grid concurrency settings for las-merging on preads in `1-preads_ovl/`

sge_option_fc <str> Grid concurrency settings for stage 2 in `2-asm-falcon/`

pa_DBdust_option <str> Passed to `DBdust`. Used only if `dust = true`.

pa_DBsplit_option <str> Passed to `DBsplit` during pre-assembly stage.

pa_HPCdaligner_option <str> Passed to `HPC.daligner` during pre-assembly stage. We will add `-H` based on''length_cutoff''.

The `-dal` option also controls the number of jobs being spawned. The number for the `-dal` option determines how many blocks are compared to each in single jobs. Having a larger number will spawn a fewer number of

larger jobs, while the opposite will give you a larger number of small jobs. This will depend on your on your compute resources available.

In this workflow, the trace point generated by `daligner` is not used. (Well, to be efficient, one should use the trace points but one have to know how to pull them out correctly first.) The `-s1000` argument makes the trace points sparse to save some disk space (not much though). We can also ignore all reads below a certain threshold by specifying a length cutoff with `-l1000`.

The biggest difference between this parameter and the `ovlp_HPCdaligner_option` parameter is that the latter needs to have a relaxed error rate switch `-e` as the alignment is being performed on uncorrected reads.

pa_dazcon_option <str> Passed to `dazcon`. Used only if `dazcon = true`.

falcon_sense_option <str> Passed to `fc_consensus`. Ignored if `dazcon = true`.

falcon_sense_skip_contained <str> Causes `-s` to be passed to `LA4Falcon`. Rarely needed.

ovlp_DBsplit_option <str> Passed to `DBsplit` during overlap stage.

ovlp_HPCdaligner_option <str> Passed to `HPC.daligner` during overlap stage.

overlap_filtering_setting <str> Passed to `fc_ovlp_filter` during assembly stage.

fc_ovlp_to_graph_option <str> Passed to `fc_ovlp_to_graph`.

skip_check <bool> If "true", then skip `LAcheck` during `LAmerge`/`LAsort`. (Actually, `LAcheck` is run, but failures are ignored.) When `daligner` bugs are finally fixed, this will be unnecessary.

dust <bool> If true, then run `DBdust` before pre-assembly.

dazcon <bool> If true, then use `dazcon` (from pbdagcon repo).

stop_all_jobs_on_failure <bool> DEPRECATED This was used for the old pypeFLOW refresh-loop, used by `run0.py`. (This is *not* the option to let jobs currently in SGE (etc) to keep running, which is still TODO.)

use_tmpdir <bool> (boolean string) whether to run each job in `TMPDIR` and copy results back to nfs If "true", use `TMPDIR`. (Actually, `tempfile.tmpdir`. See standard Python docs: https://docs.python.org/2/library/tempfile.html) If the value looks like a path, then it is used instead of `TMPDIR`.

6.2.2 fc_unzip.cfg

job_type <str> same as above. grid submission system, or "local" Supported types include: "sge", "lsf", "pbs", "torque", "slurm", "local" case-insensitive

input_fofn <str> This will be the same input file you used in your *fc_run.cfg*

input_bam_fofn <str> List of movie bam files. Only necessary if performing consensus calling step at the end.

smrt_bin <str> path to `bin` directory containing samtools, blasr, and various GenomicConsensus utilities

jobqueue <str> Queue to submit SGE jobs to.

sge_phasing <str> Phasing grid settings. Example: `-pe smp 12 -q %(jobqueue)s`

sge_quiver <str> Consensus calling grid settings. Example `-pe smp 24 -q %(jobqueue)s`

sge_track_reads <str> Read tracking grid settings. Example `-pe smp 12 -q %(jobqueue)s`

sge_blasr_aln <str> `blasr` alignment grid settings. Example `-pe smp 24 -q %(jobqueue)s`

sge_hasm <str> Final haplotyped assemble grid settings Example `-pe smp 48 -q %(jobqueue)s`

unzip_concurrent_jobs <int> Number of concurrent unzip jobs to run at a time

quiver_concurrent_jobs <int> Number of concurrent consensus calling jobs to run

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 7

Glossary

associated contig Alternate configuration (phase) of a portion of a primary contig*. See *this* discussion on primary vs associated contigs

compound path multi-paths from a single source to a single sink in a graph

contig contiguous sequence output from a genome assembler

error correction The process of combining data from multiple raw sequences with random error profile together to eliminate the errors.

full-pass subread A subread that begins at one adapter sequence and ends at another adapter sequence. A full-pass subread does not begin or end in the middle of an insert sequence.

haplotig Contig from specific haplotype

pread Pre-assembled Reads, error corrected reads through the pre-assembly process.

pre-assembly Error correction process assembling raw sequences to generate high qualityy consensus for the final step of assembly.

primary contig contig which captures a contiguous part of a genome regardless the variations due to the variation between haplotypes associated contig generated by alternative paths from a portion in the primary contig

proper overlap read overlaps without unaligned overhangs:

Quiver A highly accurate consensus and variant caller that can generate 99.999% accurate consensus sequences using local realignment and the full range of quality scores associated with Pacific Biosciences reads. Part of the SMRT® Analysis suite.

rawreads Uncorrected raw SMRTcell movie data

simple path a path without any branches in the assembly graph

string graph see The fragment assembly string graph by Eugene W. Myers, 2005

subread Each polymerase read is partitioned to form one or more subreads, which contain sequence from a single pass of a polymerase on a single strand of an insert within a SMRTbell™ template and no adapter sequences. The subreads contain the full set of quality values and kinetic measurements. Subreads are useful for applications like de novo assembly, resequencing, base modification analysis, and so on.

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 8

Frequently Asked Questions

8.1 General

8.1.1 Can I start from corrected reads?

Yes. The option *input_type* can be set to either `raw` or `preads`. In the case of the latter, fc_run.py will assume the fasta files in *input_fofn* are all error-corrected reads and it will ignore any *error correction* step and go directly into the final assembly overlapping step.

8.1.2 How do I select a length cutoff?

The option *length_cutoff* controls the read length cutoff used during the *error correction* process and *length_cutoff_pr* controls the cutoff used for the final assembly overlapping steps. In the final assembly, more reads may not lead to a better assembly due to the fact that some of the reads can be noisy and create false links in the assembly graph. Sometimes you might want to re-run the final steps of the assembly pipeline in `2-asm-falcon` with different values for `--min_len` in `run_falcon_asm.sub.sh` as this step is quick relative to the overlap detection steps in the earlier stages of the pipeline.

If you're not sure, and you are not compute resource limited, one strategy is to choose a smaller *length_cutoff* and do the computation once. Later, one can use a different *length_cutoff_pr* to achieve a more contiguous assembly

In general we recommend that you tune the cutoff so that you're left with roughly 15x to 20x for final genome assembly. If you set *length_cutoff* equal to `-1`, FALCON will attempt to autocalculate this cutoff for you.

8.1.3 What's the difference between a Primary and an Associated contig?

Primary contigs can be thought of as the longest continuous stretches of contiguously assembled sequence, while *associate contigs* can be thought of mostly as structural variants that occur over the length of the primary contigs. Thus, each alternate primary contig configuration (associated contig) can be "associated" with it's primary based on it's `XXXXXXF` prefix.

Some basic information about how the associated contigs are generated can be found in this speakerdeck (pg.14) , here (pg.14-15) and here.

Conceptually, if a genome is haploid, then all contigs should be primary contigs. However, in general there will usually still be some associated contigs generated. This is likely due to:

1. Sequencing errors
2. Segmental duplications.

For the first case, Quiver should help by filtering out low quality contigs. Since there is more sequence in the set of primary contigs for blasr to anchor reads and there is no true unique region in the erroneous associated contigs, the raw read coverage on them should be low. We can thus filter low quality *associated contig* consensus as there won't be much raw read data to support them.

For the second case, one could potentially partition the reads into different haplotype groups and construct an assembly graph for each haplotype and generate contigs accordingly.

If a genome is a diploid genome, then most of the associated contigs will be locally alternative alleles. Typically, when there are big structural variations between homologous chromosomes, there will be alternative paths in the assembly graph and the alternative paths correspond to the associated contigs. In such case, the primary contigs are "fused contigs" from both haplotypes.

FALCON_unzip is currently being developed to resolve the haplotypes so *haplotigs* can be generated. Two videos illustrating the concept - (Video 1 , Video 2)

A slide illustrating the method on a synthetic genome.

What are the differences between a_ctg.fasta and a_ctg_base.fasta

The file `a_ctg_base.fasta` contains the sequences in the primary contigs fasta that correspond to the associated contigs inside `a_ctg.fasta`. Namely, each sequence of a_ctg_base.fasta is a contiguous sub-sequence of a primary contig. For each sequence inside 'a_ctg_base.fasta, there are one or more associated contigs in a_ctg.fasta.

For a given contig in a_ctg.fa, how can I find it's primary contig map coordinates?

The 2nd field and the 3rd field of the sequence header inside `a_ctg.fa` indicate the begin node and the end node of the contig. For example, if we have a header like

```
>000000F-001-01 000941458:E 000486369:E 15593 47559 5 0.9969 0.8447
```

It means the associated contig 000000F-001-01 starts from node 000941458:E and ends at 000486369:E. Thsee two nodes should be also in the path of the corresponding primary contig. The path of the primary contig is fully specified in the file p_ctg_tiling_path, you can find exact beginning and ending points where the associated contig are attached to the primary contigs. However, the coordinates are not conserved after the Quiver consensus step, it might be necessary to do some quite alignment to recalibrate the attaching points after quiver consensus. In some case, you can even just do quick sequence alignment to find the homologous region in the primary contig of an associated contigs.

8.1.4 How does FALCON avoid chimeras given homologous repeat regions on different chromosomes?

Such repeats are typically called as "segmental duplications". Yes, Falcon will collapse these regions if the overlapper can not distinguish the repeats. As discussed above in some case, it is just like the case of a diploid genome, we can potentially resolve the two distinct haplotypes. In other cases, the repeat is more complicated, such as if there are more than 2 copies, (e.g. the middle part of contigs 4006 in page 21 of this slide deck. To resolve these regions, we'll need to do more investigation to separate the reads into more than two groups to resolve them.

8.1.5 Can Falcon handle X-ploid genome data?

Falcon, in its current form, is a "diploid or polyploid aware assembler". I believe there is no fully specific definition what a "diploid or polyploid assembler" should deliver yet at the moment of this writing. From the point of the genome assembly research field, it is still quite new. There were a couple of papers published before for diploid assemblies. However, the general strategy is the phasing adding reads on top on earlier assembly step.

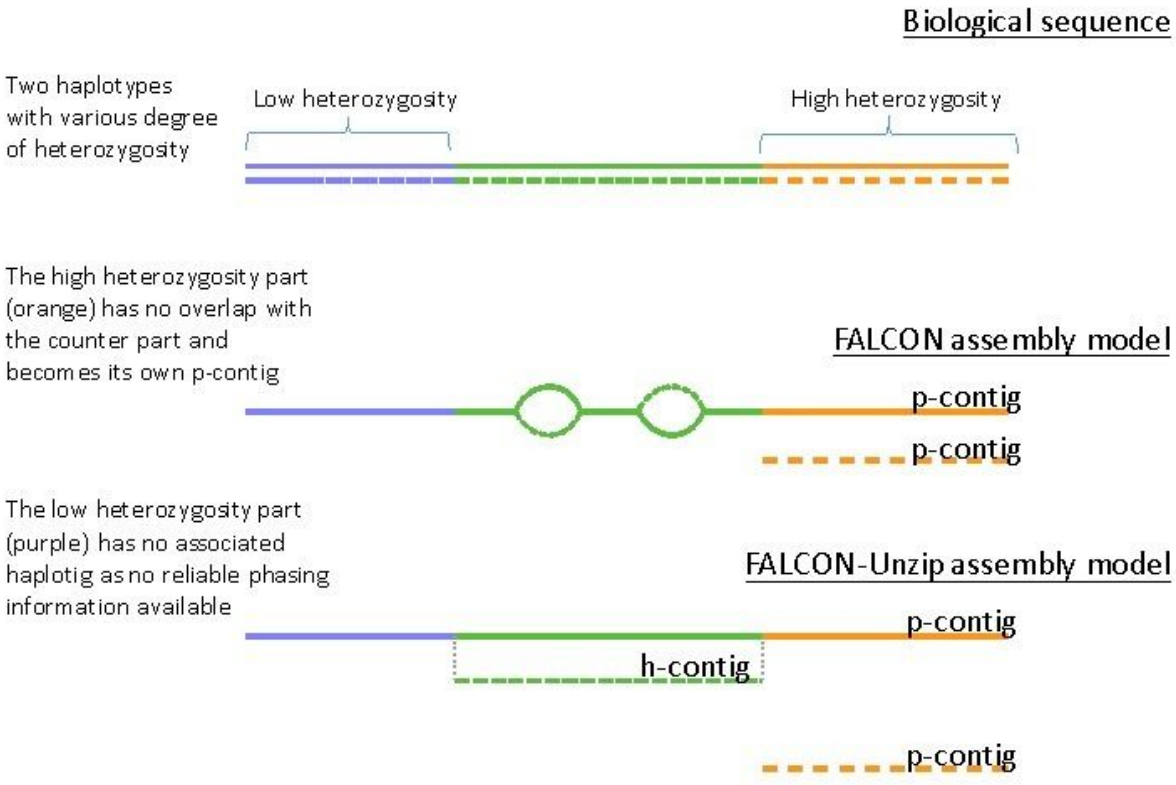
To some degree, the current Falcon assembler provides a better way to build that foundation for a full diploid / polyploid assembler. Please refer to this slide deck https://speakerdeck.com/jchin/string-graph-assembly-for-diploid-genomes-with-long-reads for some detail. Some technical details of the deck are already obsoleted for a little bit, but the general concept is still applied to most recent code in Falcon.

For a tetraploid genome, depending on the genome structure, I would argue one will get better continuity from the primary contigs if you use Falcon for assembling the genome. However, you will need to do good analysis on both primary and associated contigs (or better, the assembly graph directly) after running Falcon to interpret the results correctly. The primary contigs will be "fused" contigs from all haplotypes unless the differences between haplotypes are big such that the assembler's overlap segregate them apart already.

There are some prototype work to fully segregate the "fused primary contigs" for diploid case. I just presented the ideas in #SFAF2015 conference. For tetraploid case, it will need some hard-code non-trivial mathematics research work to get it work right.

8.1.6 Why don't I have two perfectly phased haplotypes after FALCON_unzip?

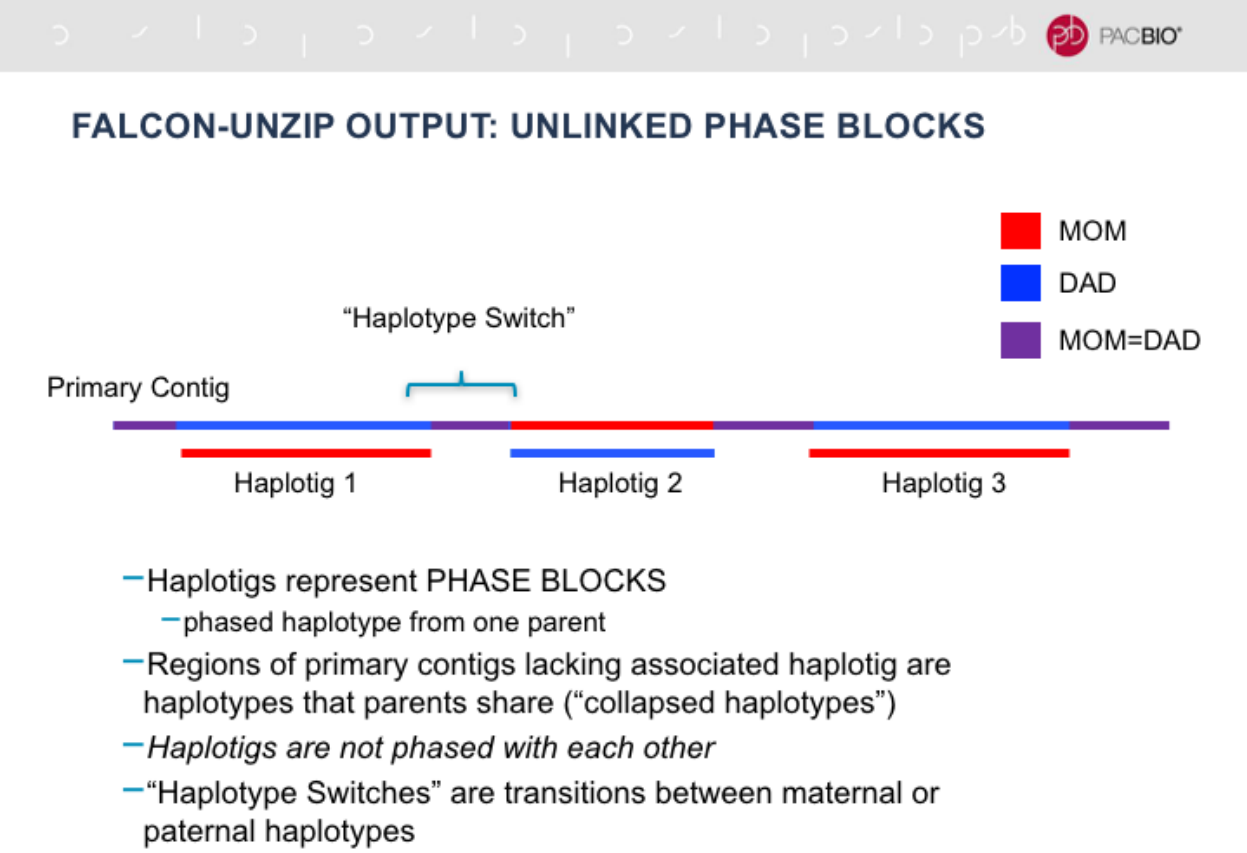
It's useful to first understand that not all genomes are alike. Haploid genomes are the holy grail of genome assembly as there is only one haplotype phase present and assembly is trivial if you have reads long enough to span repeats. Diploid and (allo/auto)polyploid genomes become difficult as there are two or more haplotype phases present. This fact, coupled with widely varying levels of heterozygosity and structural variation lead to complications during the assembly process. To understand your FALCON output, it's useful to look at this supplemental figure from the FALCON_unzip paper:



Consider the first line as a cartoon illustrating 3 ranges of heterozygosity (low/medium/high). In general, all genomes will have regions that fall into each of these three categories depending on organismal biology. During the first step of the FALCON assembly process, a diploid aware assembly graph is generated. At this point, in medium heterozygosity regions structural variation information is captured as bubbles or alternative pathways in the assembly graph whereas at high levels of heterozygosity the haplotype phases assemble into distinct primary assembly graphs.

The `FALCON_unzip` add-on module to the FALCON pipeline is an attempt to leverage the heterozygous SNP information to phase the medium level heterozygosity regions of the genome. Low heterozygosity regions have insufficient SNP density for phasing, while high heterozygosity regions will likely have already been assembled as distinct haplotypes in the primary contigs.

FALCON_unzip yields two fasta files. One containing primary contigs, and one containing haplotigs. The primary contigs fasta file is the main output that most people consider first and should consist of the majority of your genome. Primary contigs are considered `partially-phased`. What this means is that even after the unzipping process, certain regions with insufficient SNP density are unable to be phased and are thus represented as `collapsed haplotypes`. The presence of these regions of low heterozygosity makes it impossible to maintain phase across the entire primary contig. Thus primary contigs may contain phase-switches between unzipped regions. The haplotigs file will consist of the `unzippapble` or `phaseable` regions of the genome and are considered fully phased. This means there should be no phase switching within a haplotig and each haplotig should represent only one phase. See this figure for reference:



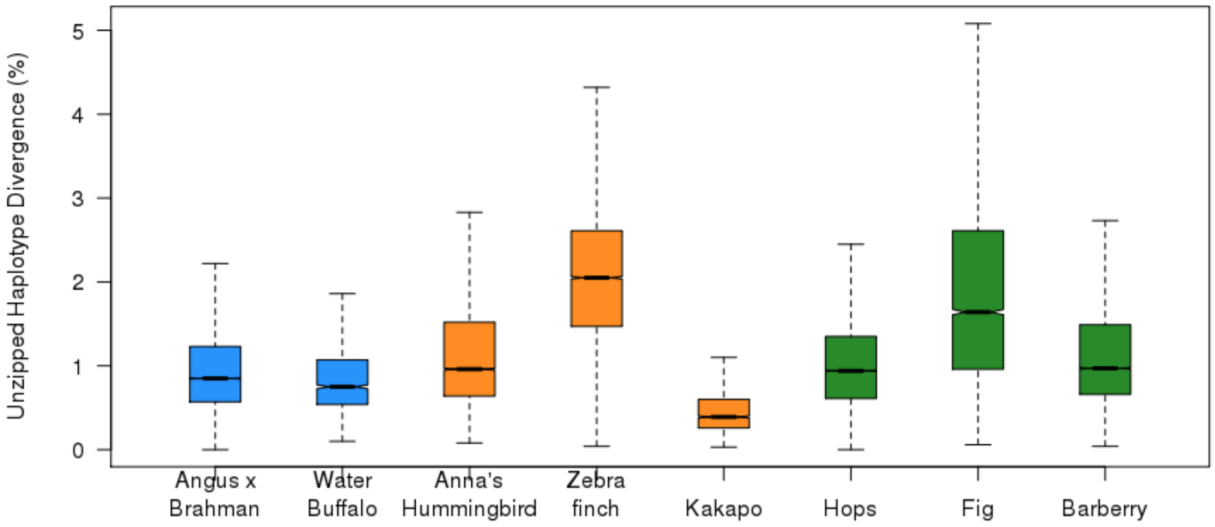
It's also important to note that in high heterozygosity situations, we often see the primary contig fasta file approaching 1.5X+ the expected haploid genome size, due to the assembly of both phases of certain chromosomes or chromosomal regions in the primary assembly.

Also, one needs to consider that FALCON_unzip was designed to phase the plant and fungal genomes in the 2016 Nature Methods paper above, but many people have successfully used it to help phase their genome of interest. But as always with free software on the internet, your mileage may vary.

8.1.7 How much haplotype divergence can FALCON-Unzip handle?

The magnitude of haplotype divergence determines the structure of the resulting FALCON-Unzip assembly. Genomic regions with low heterozygisty will be assembled as collapsed haplotype on a single primary contig. Haplotypes up to ~5% diverged will be Unzipped, while highly divergent haplotypes will be assembled on different primary contigs. In the latter case, it is up to the user to identify these contigs as homologous using gene annotation or sequence alignment.

For a variety of FALCON-Unzip assemblies, here is the distribution of haplotype divergence for unzipped regions. Each haplotig was aligned to the corresponding primary contig with nucmer, filtered with delta-filter and divergence was estimated with show-choords. (Data credits to John Williams, Tim Smith, Paolo Ajmone-Marsan, David Hume, Erich Jarvis, John Henning, Dave Hendrix, Carlos Machado, and Iago Hale).



8.1.8 Why does FALCON have trouble assembling my amplicon data?

FALCON was designed for whole genome shot gun assembly rather than amplicon assembly. In whole genome shot-gun assembly we suppress repetitive high copy regions to assemble less repetitive regions first. When you assemble PCR product of a short region in a genome, FALCON sees the whole thing as a high copy repeat and filters alot of the data out.

You can try to down sample your data and make the daligner block size even smaller (reduce -s50 in pa_DBsplit_option and ovlp_concurrent_jobs) and increase the overlap filter thresholds (–max_diff 100 –max_cov 100 in overlap_filtering_setting) to try to make it work, however it's not really within the scope of the FALCON algorithm.

8.2 Workflow

8.2.1 How do I restart a failed workflow?

Often times restarting a FALCON job due to unexplained failure is an easy process. It's typically just a matter of removing any sentinel files and re-invoking `fc_run.py fc_run.cfg` from the FALCON root directory. Read this section of the wiki for details. If your job failed due to quota or other disk full issues , you may need to wipe the directory and start over again due to corrupt DB's

8.2.2 How do I turn on logging?

See this github issue.

In short, you should pass `logging.ini` as the 2nd argument to *fc_run.py*

```
$ fc_run.py fc_run.cfg logging.ini
```

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 9

Changelog

9.1 3/12/2018

9.2 8/08/2018

9.2.1 FALCON Updates:

Repeat Masking

- Integration of DAMASKER: Tandem repeat masking (done) and general repeat masking (in progress)

Improved default settings for microbial assembly

- Use one longest read per ZMW: reduced chimerism, coverage bias
- Retuned parameters to increase contiguity

New! GFA-2 support

- Assembly graphs now written in both GFA-1 and GFA-2 formats
- Placement coordinates of associate contigs now available in a new "contig.gfa2" file

Performance Improvements

- General workflow and resource specification improvements
- Easier integration of future features with Pbsmrtpipe

9.2.2 FALCON-Unzip Updates:

Improved Haplotig Extraction

- Algorithm and data structure improvements reduce haplotype switching and improve extraction (resolved nested and overlapping haplotigs)

- Can now handle circular contigs!

New! Placement Files

- Haplotig placement (PAF format) generated after Unzip
- Easier integration with FALCON-Phase

Performance Improvements

- Use of Minimap2 instead of BLASR for phasing in Unzip reduces time requirements
- Significantly reduced memory consumption of the final stage of Unzip (preads no longer have to be loaded in memory)
- Unzipping and polishing are now combined in the same workflow and run consecutively.

CHAPTER 10

Relevant Talks and Poster Presentations

10.1 Sequencing, Finishing and Analysis in the Future

May 23rd 2018, Sante Fe, NM

10.2 *FALCON-Phase talk by Sarah Kingan*

`FALCON-Phase`

10.3 SMRT Informatics Developers Conference

Jan 17th 2018, San Diego, CA

`Agenda`

10.4 *Keynote by Tim Smith*

`TimSmith_The_evolution_of_reference_assembly`

10.5 *Software Updates from PacBio*

`JimDrakePacBio_SMRTAnalysisUpdates`

10.6 *Lightning Talks*

ZevKronenberg_PhaseGenomics_HighQualityGenomesPacBioHiC

SKoren_NHGRI_TrioBinning

ETseng_PacBio_IsoPhase

AConesa_UF-PF_IsoSeq_SQUANTI_TAPPAS

BBowman_PacBio_RepeatAnalysisNoAmp

AWenger_PacBio_PBSV

10.7 PAGXXVI in San Diego, January 2018

10.8 *PacBio Presented Posters*

Concepcion-PAG-2018-Maize_Soy_SV.pdf

Concepcion-PAG-2018-S.californicum_de novo_assembly.pdf

Kingan-PAG-2018-BestPracticesforDiploidAssemblyofComplexGenomesUsingPacBioHops.pdf

Tseng-PAG-2018-Haplotyping-of-full-length-transcripts.pdf

10.9 *Talks*

Zev Kronenberg (Phase Genomics, fmr Eichler Lab, UW) work on great ape genomes ZKronenberg_GreatApeComparativeGenomics_PAG2018.pdf

Lloyd Low (Williams Lab, Adelaide) work on waterbuffalo, includes issue of haplotype switching and scaffolding LLow_WaterBuffalo_PAG2018.pdf

10.10 Tools for Polyploids Meeting

San Diego Botanical Garden, Jan 12th, 2018

KinganPacBio_ToolsForPolyploidsPAG2018.pdf

10.11 PacBio East Coast User Group Meeting

Baltimore, June 27th, 2017

Kingan_DiploidGenome_ECUGM2017_BFX.pdf

Fritz_Sedlazeck_SVswithPacB.pdf

Ghurye_PacBHiC_Asm.pdf

Wenger_pbsv_BFXwrkshop.pdf

Goodwin_SamplePrep_ExtraLongLibraries.pdf

Smith_SamplePrep_BestPracLargeInsertLib.pdf

HamidAshrafi_BlueberryAsmIsoSeq.pdf

Caution: These documents refer to an obsolete way of installing and running FALCON. They will remain up for historical context and for individuals still using the older version of FALCON/FALCON_unzip.

Attention: The current PacBio Assembly suite documentation which includes new bioconda instructions for installing FALCON, FALCON_unzip and their associated dependencies can be found here pb_assembly

CHAPTER 11

Running Unzip on HGAP4 output

11.1 Overview

HGAP4 is a FALCON-based assembly pipeline, available through the SMRT Link interface. The pipeline itself encapsulates *de novo* assembly and polishing of the resulting contigs, but *not* the FALCON-unzip process as well. FALCON-unzip is currently available as a standalone tool, runnable only via command line.

Although HGAP4 runs FALCON under the hood, the folder structure it generates is different than that of FALCON. The FALCON-unzip, however, requires the assembly folders to be formatted in the FALCON-style.

This tutorial describes the necessary steps required to adjust the HGAP4 output to be compatible with a form required by FALCON-unzip.

In brief, the majority of work required to adjust the HGAP4 output to a FALCON-compatible directory structure is implemented in a script called hgap4_adapt. This script lives in the FALCON repository.

The complete process is composed of the following steps:

1. Installing FALCON and FALCON-unzip.
2. Running hgap4_adapt.
3. Creating the fc_unzip.cfg configuration file for FALCON-unzip.
4. Creating the input.fofn and input_bam.fofn.
5. Running FALCON-unzip.

IMPORTANT: FALCON-unzip can only be run on HGAP4 jobs which had the Save Output for Unzip option turned on. It is not possible to run FALCON-unzip otherwise, because critical files will be missing from your job's output.

11.2 1. Installing FALCON and FALCON-unzip

The latest versions of FALCON and FALCON-unzip are available as precompiled Linux binaries. The easiest approach to installing them is through a wrapper script, described here:

Quick Start

Follow this approach to set-up the environment before moving on to step 2.

Alternatively, one can install the binaries manually by following the instructions here: https://github.com/PacificBiosciences/FALCON_unzip/wiki/Binaries

11.3 2. Running `hgap4_adapt`

Once the `FALCON` installation was successful, one needs to activate the installation environment to make the `hgap4_adapt` script available. This will also activate `FALCON` and `FALCON-unzip`. To verify the installation, run the following:

```
source /path/to/your/install/dir/fc_env/bin/activate
python -m falcon_kit.mains.hgap4_adapt --help
```

If everything was successful, this should output verbose usage information to screen. After this is set-up and working, adapting an existing HGAP4 run is as simple as the following example (take note of the dummy path, and replace it with a real one):

```
source /path/to/your/install/dir/fc_env/bin/activate

job_dir=/path/to/your/hgap4/job/123/123456/
mkdir -p example1
cd example1
python -m falcon_kit.mains.hgap4_adapt --job-output-dir=${job_dir}
```

The result should be visible in the `example1` directory - it should now be populated to folders resembling a typical `FALCON` assembly run.

11.4 3. Creating the `fc_unzip.cfg` configuration file for `FALCON-unzip`

For help on .cfg files, please take a look at these Wiki pages:

- https://github.com/PacificBiosciences/FALCON/wiki
- https://github.com/PacificBiosciences/FALCON_unzip/wiki
- https://github.com/PacificBiosciences/FALCON-integrate/wiki

11.5 4. Creating the `input.fofn` and `input_bam.fofn`

The `input.fofn` file ("file of file names") contains the paths to files containing plain FASTA sequences of your raw reads, one file per row. All raw reads in the FASTA format should be available in your job dir:

```
job_dir=/path/to/your/hgap4/job/123/123456/
mkdir -p example1
cd example1

echo "${job_dir}/tasks/pbcoretools.tasks.gather_fasta-1/file.fasta" > input.fofn
```

The `input_bam.fofn` is required for the polishing step. This file is composed of a list of all BAM files from the input dataset which was provided to the initial HGAP4 run:

```
source /path/to/your/install/dir/fc_env/bin/activate

job_dir=/path/to/your/hgap4/job/123/123456/
mkdir -p example1
cd example1

dataset summarize ${job_dir}/tasks/pbcoretools.tasks.filterdataset-0/filtered.
↪subreadset.xml | grep -E "*.bam$" > input_bam.fofn
```

11.6 5. Running `FALCON-unzip`

Before running `FALCON-unzip`, the adapted folder structure should be similar to the following:

```
$ cd example1
$ ls | xargs -n 1
    0-rawreads
    1-preads_ovl
    2-asm-falcon
    fc_unzip.cfg
    input_bam.fofn
    input.fofn
```

Finally, to run `FALCON-unzip`, do the following:

```
source /path/to/your/install/dir/fc_env/bin/activate
cd example1
fc_unzip.py fc_unzip.cfg
fc_quiver.py fc_unzip.cfg
```

FALCON is a diploid aware genome assembler designed for Pacific Biosciences long read data.

- *About* - About FALCON
- *Quick Start* - Want to start using FALCON immediately?
- *Tutorial* - Follow an example on how to run FALCON
- *Pipeline* - How does a FALCON job work?
- *Commands* - Information on the different commands used
- *Parameters* - Descriptions of parameters found in fc_run.cfg
- *Glossary* - Glossary of FALCON jargon
- *Frequently Asked Questions* - Frequently asked questions
- *Changelog* - Changes
- *Resources* - Talks and slides about PacBio
- *Running Unzip on HGAP4 output* - How to run apply Unzip on finished HGAP4 jobs?

Index