Metagenomics Processing

Instructions for processing high-throughput shotgun metagenomics reads

Note - Familiarize yourself with the steps in the workflow using the training dataset provided in /srv/data/training/metagenomics/shakya2013/ prior to following the workflow for the first time on real data.

Contents

References

Boisvert, S., Raymond, F., Godzaridis, E., Laviolette, F., Corbeil, J. (2012) Ray Meta: scalable *de novo* metagenome assembly and profiling. *Genome Biology*, 13: R122. doi: 10.1186/gb-2012-13-12-r122.

Clark, S. C., Egan, R., Frazier, P. I., Wang, Z. (2012) ALE: a generic assembly likelihood evaluation framework for assessing the accuracy of genome and metagenome assemblies. *Bioinformatics*, 29(4): 435-443. doi: 10.1093/bioinformatics/bts723.

Crusoe, M. R., Alameldin, H. F., Awad, S., et al. (2015) The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Research*, 4:900. doi: 10.12688/f1000research.6924.1.

Ghodsi, M., et al. (2013) *De novo* likelihood-based measures for comparing genome assemblies. *BMC Research Notes*. 6:334. doi: 10.1186/1756-0500-6-334.

Gomez-Alvarez, V., Teal, T. K., Schmidt, T., M. (2009) Systematic artifacts in metagenomes from complex microbial communities. *ISME J*, 3(11):1314–7. doi:10.1038/ismej.2009.72.

Haider, B., Ahn, T. H., Bushnell, B., Chai, J., Copeland, A., Pan, C. (2014) Omega: an overlap-graph de novo assembler for metagenomics. *Bioinformatics*, 30(19): 2717-2722. doi: 10.1093/bioinformatics/btu395.

Langmead, B., Salzberg, S. (2012) Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9: 357-359. doi: 10.1038/nmeth.1923.

Li, D., Liu, C-M., Luo, R., Sadakane, K., and Lam, T-W. (2015) MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*. doi: 10.1093/bioinformatics/btv033

Li, D., Luo, R., Liu, C.M., Leung, C.M., Ting, H.F., Sadakane, K., Yamashita, H. and Lam, T.W. (2016) MEGAHIT v1.0: A Fast and Scalable Metagenome Assembler driven by Advanced Methodologies and Community Practices. Methods.

Li, H., et al. (2009) The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16): 2078-9. doi: 10.1093/bioinformatics/btp352.

Martin, M. (2011) Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet.journal*, 17(1): 10-12. doi: 10.14806/ej.17.1.200.

Mikheenko, A., Saveliev, V., Gurevich, A. (2015) MetaQUAST: evaluation of metagenome assemblies. *Bioinformatics*. doi: 10.1093/bioinformatics/btv697.

Peng, Y., Leung, H. C., Yiu, S. M., Chin, F. Y. (2012) IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28(11): 1420-1428. doi: 10.1093/bioinformatics/bts174.

Quinlan, A. R. and Hall, I. M. (2010) BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*. 26(6): 841–842. doi: 10.1093/bioinformatics/btq033.

Data Preparation

Starting Materials:
- Raw HTS paired-end reads in FASTQ format

Initialize recurring variables:

```
raw_data="/srv/data/mg/projects/<project_code>/short_reads"
project="~/projects/<project_code>"
results="${project}/results"
scripts="${project}/src"
data="${project}/data"
figures="${results}/figures"
assembly="${results}/assembly"
analysis="${results}/analysis"
logs="${results}/logs"
qc="${results}/qc"
stats="${results}/stats"
cpus=<number_cpus>
sample="<sample_name>"
```

* environmental variables will need to be reset for each new terminal session. If working on multiple projects within the same session, alter the variable names to be specific to a given project.
* a modules file containing the above variable names can be loaded to facilitate variable assignment.

Make a new project directory and link the appropriate raw data files to it:

```
mkdir --parents ${scripts} ${data} ${stats} ${figures} ${assembly} ${analysis} ${logs} ${qc}
ln -svi ${raw_data}/* ${data}
```

(if applicable) Combine multiple metagenomic datasets together:

```
combined="<unique_name>"

find ${project}/data/ -regex "<pattern>" | sort | xargs -d "\n" zcat | gzip - > ${combined}.<strand>.fastq.gz &
```

* do separately for both forward and reverse reads if the reads are not already interleaved.
* replace **<pattern>** with the appropriate string that identifies forward reads from reverse reads. Typically this will be either `".*_R1_.*"` and `".*_R2_.*"` or `".*\.forward\..*"` and `".*\.reverse\..*"`.
* this **should not** be done unless there is good reason to pool metagenomes prior to quality control, and only for samples that are expected to have a similar error profile (i.e samples sequenced on the same lane).
* replace all instances of `${sample}` with `${combined}` in subsequent steps.

Generate and review quality control report on reads:

```
forward="${data}/${sample}.01.forward.fastq.gz"
reverse="${data}/${sample}.01.reverse.fastq.gz"

srun fastqc --noextract --outdir ${stats} ${forward} ${reverse}
```

or

```
srun --x11=first fastqc
```

* FastQC can be run in either interactive or batch mode. If run in interactive mode, X11 forwarding must be enabled at the time of connection to the server (ssh -X). If run in batch mode, copy the output html files onto your local machine and open them in a browser to view the results.

Preprocessing

Preprocessing of the sequencing data can be done sequentially or all at once. If unfamiliar with the workflow, run through it step by step until comfortable with the various parameters used in each program. Once you have a good understanding of what is being done to the data, you are welcome to try running everything in one command.

Step by step

Trim adapters from reads:

```
srun --cpus-per-task ${cpus} bbduk.sh -Xmx10g ziplevel=9 threads=${cpus} qin=33 ref=/srv/databases/contaminants/truseq_adapters.fa in1=${forward} in2=${reverse} out=${qc}/${sample}.interleaved.atrim.fq.gz stats=${stats}/${sample}.adapter_stats.txt ftm=5 ktrim=r k=23 mink=9 rcomp=t hdist=2 tbo tpe minlength=0 2>${logs}/${sample}.adapters.log &
```

* when a template lacks enough bases to sequence, the sequencer will read through the adapter. This happens when there are fragments in the library that are shorter than the length of a read, and will result in the presence of an adapter at the 3'-end of the sequence.
* a contaminant might appear in the 5'-end of a sequence if two adapters were ligated during the library preparation stage, possibly as a consequence of high adapter to sample DNA ratio, and the resulting fragment sequenced along with sample DNA.
* this step assumes that the NebNext adapter for Illumina, TruSeq Universal primers, and Indexed primers were used during library preparation. If not, a fasta file containing the appropriate adapter/primer sequences will need to be provided instead. The FASTQC report is a good source of information for identifying what adapters, if any, are present in the dataset. However the FASTQC database of adapters is not comprehensive, so some could easily be missed. A manual search through the sequencing data is the only sure way to find artifactual sequences in the data. The linux utility `less` provides an efficient way to scan a file for a specific string of characters (i.e. adapter sequences).

Discard contaminant reads:

```
mkdir ${qc}/discarded


srun --cpus-per-task ${cpus} bbduk.sh -Xmx10g threads=${cpus} qin=33 interleaved=t
ref=/srv/databases/contaminants/phix174.fa.gz
in=${qc}/${sample}.interleaved.atrim.fq.gz
out=${qc}/${sample}.interleaved.atrim.decontam.fq.gz
outm=${qc}/discarded/${sample}.phix.fq.gz k=31 hdist=1 mcf=0.9
stats=${stats}/${sample}.phix_stats.txt 2>${logs}/${sample}.phix.log &
```

* DNA from the genome of bacteriophage PhiX174 is often used as a spike-in control during Illumina sequencing runs and should be removed when present.
* this step can also be used to remove other potential contaminating sequences, such as host DNA - simply provide `ref` a comma-separated list of FASTA files containing reference sequences.

Note - Trimming based on quality is often performed on HTS reads because sequences containing erroneous bases can create corresponding errors in an assembly. Sequencing error can also inflate the complexity of the graph structure used in short-read assembly, increasing both the resources that an assembler consumes and the amount of time it takes to run. Vigorous trimming will remove much of this error; however, it can also result in substantial data loss, potentially hurting the detection capability of the study. Therefore the optimal strength of read trimming will likely vary project-to-project, and will depend on factors such as sequencing depth and research goals. In a study using publically available Illumina paired-end transcriptome data, MacManes (2014) found that overly stringent trimming resulted in worse assemblies according to the majority of metrics measured, particularly for low coverage datasets. There was also bias observed in the portion of the assembly missing from the trimmed data, with the transcripts lost from the aggressively trimmed dataset identified as being the more lowly expressed ones. The negative effects of trimming were reduced, however, as coverage

was increased. Based on these findings, the author recommends the use of a gentle trimming strategy (Phred quality score between **2** and **5**) except under specific scenarios when more aggressive trimming is warranted, for example, when examination of community microdiversity (e.g. SNP detection) is the study aim or when sequencing depth is very high. The applicability of these guidelines to metagenomics was investigated on the Shakya et al. (2013) dataset, and the results were largely in agreement with MacManes (2014). Aggressive trimming was observed to only modestly improve assembly for high coverage datasets, while having an increasingly detrimental impact as coverage was reduced. As coverage appears to be one of the more important determinants in regards to the extent that trimming strategy affects assembly quality, a promising avenue for selection of an appropriate quality threshold would be to make use of a program like Nonpareil to obtain an estimate of the metagenome's average coverage.

(optional) Estimate the metagenome's average coverage and compute an accumulation curve:

```
srun readstats.py --csv --output ${stats}/${sample}.atrim.decontam.readstats.csv
${qc}/${sample}.interleaved.atrim.decontam.fq.gz 2>/dev/null &

nreads=$(expr $(tail -n 1 ${stats}/${sample}.atrim.decontam.readstats.csv | awk -F
"\"*,\"*" '{ print $2 }') \* 5 / 100)  #should be around 1-5% of total dataset size
echo $nreads
minlen="<minimum read length>"  #can be same as that which will be used to trim based on quality

srun sample-reads-randomly.py --num_reads ${nreads} --output /dev/stdout
${qc}/${sample}.interleaved.atrim.decontam.fq.gz 2>/dev/null | srun --cpus-per-task
$(if [[ $cpus > 4 ]]; then echo 4; else echo "$(( ${cpus} - 1 ))"; fi) qtrim --threads
$(if [[ $cpus > 4 ]]; then echo 3; else echo "$(( ${cpus} - 2 ))"; fi) --interleaved
--qual-offset 33 --trunc-n --min-len ${minlen} --leading 20 --trailing 20
--sliding-window 4:20 -o ${qc}/${sample}.forward.subset.fq -v
${qc}/${sample}.reverse.subset.fq /dev/stdin 2>${logs}/${sample}.subset.qtrim.log &

srun --cpus-per-task ${cpus} nonpareil -t ${cpus} -f fastq -T kmer -k 32 -x 0.2 -s
${qc}/${sample}.forward.subset.fq -b ${stats}/${sample}.forward.cov &
srun --cpus-per-task ${cpus} nonpareil -t ${cpus} -f fastq -T kmer -k 32 -x 0.2 -s
${qc}/${sample}.reverse.subset.fq -b ${stats}/${sample}.reverse.cov &

rm ${qc}/${sample}.forward.subset.fq ${qc}/${sample}.reverse.subset.fq
cd ${stats}

srun --pty R
library(Nonpareil)

svg('~/projects/<project_code>/results/figures/<sample_name>.np_curve.svg', height=7,
width=7)
ncurve <- Nonpareil.curve.batch(c('<sample_name>.forward.cov.npo',
'<sample_name>.reverse.cov.npo'), libnames=c('<sample_name>_forward',
'<sample_name>_reverse'))
Nonpareil.legend('bottomright')
```

```
dev.off()

ncurve[, 'LRstar']

quit()

cd ${project}
```
* compare the dataset's actual size in base pairs (first column in the readstats report) to the estimated sequencing effort required to reach an average coverage of 95% (LR*). If LR* is less than the dataset size, a larger threshold (Phred >= 10) can be used during quality trimming.
* this step only needs to be performed on a sample or two if processing multiple metagenomes in a project and the sequencing effort and community complexity is expected to be similar for all. Simply extrapolate the conclusion to the remaining samples.

Note - In shotgun genomics/metagenomics datasets, the respective frequency of a nucleotide at any given position should remain fairly constant. But sometimes bases near the beginning or end of the nucleotide frequency distribution (visualized as a base composition histogram in the FASTQC report) will exhibit unusually high or low frequencies relative to other positions; this could be an artifact of the sequencing process (i.e. remaining adapter content) or the result of non-random fragmentation of DNA during library preparation. If the former, these are erroneous bases and should be cropped off. However, if there are many such positions, more aggressive adapter trimming in place of non-discriminative cropping will preserve a larger number of correct bases. Therefore only **one or two** bases at most should be trimmed from either end, regardless of base composition skew.

Generate a base composition histogram for the adapter-trimmed reads:
```
srun --cpus-per-task ${cpus} bbduk.sh -Xmx10g threads=${cpus} qin=33 interleaved=t
in=${qc}/${sample}.interleaved.atrim.decontam.fq.gz
bhist=${stats}/${sample}.base_frequencies.hist &

cd ${stats}

srun --pty R
library(ggplot2)
rlength <- <max read length>

bhist <- data.frame(read.table("<sample>.base_frequencies.hist", sep="\t",
row.names=1), strand=rep(c("Forward", "Reverse"), times=c(rlength, rlength)),
base=rep(0:(rlength-1), times=2))
colnames(bhist) <- c("A", "C", "G", "T", "N", "strand", "base")
bhist <- data.frame(bhist[,c("base", "strand")], stack(bhist, select=c("A", "C", "G",
"T", "N")))

svg("~/projects/<project_code>/results/figures/<sample>.base_frequencies.svg",
height=6, width=9)
```

```
ggplot(bhist, aes(x=base, y=values, color=ind)) + geom_line() + facet_grid(~strand) +
xlab("Base Position") + ylab("Frequency") + theme(legend.title=element_blank())
dev.off()

quit()

cd ${project}
```

(optional, if applicable) Determine the cause of the skewed nucleotide frequency distribution:

```
nreads=$(expr $(tail -n 1 ${stats}/${sample}.atrim.decontam.readstats.csv | awk -F
"\"*,\"*" '{ print $2 }') \* 15 / 100)

srun --cpus-per-task ${cpus} tadpole.sh -Xmx20g threads=${cpus} interleaved=t
mode=contig minprob=0.8 k=31 reads=${nreads}
in=${qc}/${sample}.interleaved.atrim.decontam.fq.gz out=${qc}/${sample}.quick_assem.fa
2>${logs}/${sample}.quick_assem.log &

srun --cpus-per-task ${cpus} bbmap.sh threads=${cpus} nodisk=t interleaved=t
reads=${nreads} in=${qc}/${sample}.interleaved.atrim.decontam.fq.gz
ref=${qc}/${sample}.quick_assem.fa mhist=${stats}/${sample}.mapping_errors.hist &

rm ${qc}/${sample}.quick_assem.fa
cd ${stats}

srun --pty R
library(ggplot2)

mhist <- read.table("<sample>.mapping_errors.hist", sep="\t")
colnames(mhist) <- c("Base", "MatchForward", "SubForward", "DelForward", "InsForward",
"NForward", "OtherForward", "MatchReverse", "SubReverse", "DelReverse","InsReverse",
"NReverse", "OtherReverse")
mhist <- data.frame(Base=mhist[,"Base"], stack(mhist, select=c("SubForward",
"DelForward", "InsForward", "SubReverse", "DelReverse","InsReverse")))

svg("~/projects/<project_code>/results/figures/<sample>.mapping_errors.svg", height=6,
width=8)
ggplot(mhist, aes(x=Base, y=values, color=ind)) + geom_line() + xlab("Base Position")
+ ylab("Mapping Error Rate") + theme(legend.title=element_blank())
dev.off()

quit()

cd ${project}
```

* a substantially higher mapping error rate in the skewed bases relative to the rest of the read content would suggest that contaminants or sequencing bias is artificially inflating the frequency of some bases at these positions. If this is not the case, then the bias observed is likely due to

non-random fragmentation and the bases should be kept as they are legitimate representatives of template DNA.
* if processing more than one metagenome in a project, this step only needs to be performed once per run. For example, if eight multiplexed samples were sequenced together, the cause of any deviation from uniformity detected in one sample's nucleotide frequency distribution would be the same cause of skew in the other seven samples. Therefore, it is only necessary to determine this for a single sample.

Trim reads based on quality score and filter by length:

```
qscore="<quality score threshold>"
window="<sliding window size>"

srun --cpus-per-task $(if (( $cpus > 1 && $cpus <= 4 )); then echo ${cpus}; else echo
4; fi) qtrim --threads $(if (( $cpus > 1 && $cpus <= 4 )); then value=$(( $cpus - 1
)); echo $value; else echo 3; fi) --interleaved --qual-offset 33 --min-len ${minlen}
--crop <Fbases>,<Rbases> --headcrop <Fbases>,<Rbases> --leading ${qscore} --trailing
${qscore} --sliding-window ${window}:${qscore} -o
${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz -s
${qc}/${sample}.singles.atrim.decontam.qtrim.fq.gz
${qc}/${sample}.interleaved.atrim.decontam.fq.gz 2>${logs}/${sample}.qtrim.log &
```

* for Illumina data, quality scores are related to base-call error probabilities by the logarithmic function Q = -10 * $\log_{10}$(P). A quality score of 20 means that there is a one in one-hundred chance that the base was called incorrectly, whereas a quality score of 10 means that there is a one in ten chance that the base was called incorrectly.
* recent Illumina technologies encode quality scores with the CASAVA >= 1.8 pipeline (phred33).
* starting at the 5’-end, the sliding-window will move along the sequence `window_size` bases at a time. When the average quality score falls below the threshold, the algorithm will search the bases immediately preceding the current window and the bases at the start of the window to locate the exact position where the drop in quality first occurred; the read will then be trimmed at that position. Use a shorter window size in combination with a low quality threshold (i.e 4-6:2-10) and a longer window size in combination with a high quality threshold (i.e. 6-10:10-30).
* the leading and trailing parameters of `qtrim` will discard bases from either end of a sequence when the associated quality score falls below the provided threshold. These steps will be performed prior to the sliding-window.
* the `headcrop` and `crop` parameters specify the exact number of bases to be removed from the start and end of all sequences in the dataset. The arguments allow for different values to be provided for the forward and reverse reads, respectively, by separating them with a comma with no spaces (e.g. `--headcrop 1,2`).
* there is little consensus in the literature on what the minimum length should be to retain a sequence after quality control. One simple strategy often used is to define a hard cutoff value - discarding a sequence when the length falls below a threshold, typically in the range of 50 and 75 bp for read lengths produced by a modern Illumina platform. An alternative strategy is to

base this decision on properties of the dataset itself. For instance, MG-RAST will remove a sequence if its length is less than a user defined number of standard deviations from the average. Creators of HUMAnN, on the other hand, recommend removing reads with lengths less than 75% of the original sequence. Notice that all of these strategies require the user to specify an entirely arbitrary value, so do what you think is best. But for whatever value you use, be aware that it will impose a ceiling on the largest k-mer size that can be used for assembly.
* qtrim is largely io limited and should not be used with more than four cpus per job.

All at once

Discard contaminants, trim adapters, and quality trim reads:

```
qscore="<quality score threshold>"
window="<sliding window size>"
minlen="<minimum read length>"

mkdir ${qc}/discarded

srun --cpus-per-task $(t=(expr ${cpus} - 2); if [[ t -lt 1 ]]; then echo 1; else echo
$t; fi) bbduk.sh -Xmx10g ziplevel=9 threads=$(t=(expr ${cpus} - 2); if [[ t -lt 1 ]];
then echo 1; else echo $t; fi) qin=33
ref=/srv/databases/contaminants/truseq_adapters.fa in1=${forward} in2=${reverse}
out=stdout.fq stats=${stats}/${sample}.adapter_stats.txt ftm=5 ktrim=r k=23 mink=9
rcomp=t hdist=2 tbo tpe minlength=0 2>${logs}/${sample}.adapters.log | srun bbduk.sh
-Xmx20g threads=1 qin=33 interleaved=t in=stdin.fq out=stdout.fq
outm=${qc}/discarded/${sample}.phix.fq.gz
ref=/srv/databases/contaminants/phix174.fa.gz k=31 hdist=1 mcf=0.9
stats=${stats}/${sample}.phix_stats.txt 2>${logs}/${sample}.phix.log | srun qtrim
--threads 1 --interleaved --qual-offset 33 --min-len ${minlen} --crop
<Fbases>,<Rbases> --headcrop <Fbases>,<Rbases> --leading ${qscore} --trailing
${qscore} --sliding-window ${window}:${qscore} -s
${qc}/${sample}.singles.atrim.decontam.qtrim.fq.gz -o
${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz /dev/stdin
2>${logs}/${sample}.qtrim.log
```

Final steps

Check the quality of the remaining reads to see if additional quality control is needed:

```
mkdir qc_stats
srun fastqc --noextract --outdir qc_stats
${sample}.interleaved.atrim.decontam.qtrim.fq.gz
```

Repeat preprocessing steps with a different size insert library, if applicable.

Assembly

A number of modern, reasonably accurate de Bruijn graph-based options are available to choose from for assembling metagenomes from short-length HTS reads, including Ray Meta, IDBA-UD, MEGAHIT, and SPAdes/MetaSPAdes. The choice of assembler can depend on multiple factors such as available computational resources, project aim, level of bioinformatics expertise, and program flexibility. Several comparative reviews of popular assembly software have recently been published which can serve as guides for selecting the right assembler for a project. A good place to start is van der Walt et al. (2017), which features a decision tree for quick determination of the most appropriate assembler for a given scenario. Tests run on some of the lab's metagenomes largely recapitulate the conclusions reached by Vollmers et al. (2017), who recommend different assemblers for different tasks: MEGAHIT for capturing the full diversity of the community or when memory is a limiting factor, Ray Meta for genome reconstruction of high abundance community members, and MetaSPAdes for general use if sensitivity to microdiversity is not required. An interesting take on assembler performance was given in Awad et al. (2017), where a comparative analysis was done on the performance of three multi-kmer assemblers - IDBA-UD, MetaSPAdes, and MegaHit - using a mock dataset composed of ~64 strains of bacteria and archaea. The author's concluded that, in general, MEGAHIT produced the most accurate assemblies. But perhaps the more interesting finding of the study was that strain level variation was shown to be a major confounder of metagenome assembly, regardless of choice of assembler, with considerable assembly loss observed in the presence of closely related strains (defined as 2% Jaccard similarity or higher in minhash signature); this "strain confusion" can have important implications for pooled assembly, as pooling metagenomes can result in increased strain heterogeneity. The lab's PUMA project is currently developing recommendations for pooling samples prior to assembly.

As HTS technologies improve and read lengths get longer, the overlap-graph (OLC and its successor string-graph) approach to sequence assembly is becoming more feasible. One option for assembling metagenomics reads of moderate length and high read depth that follows the string graph approach is DISCO. DISCO might be worth trying if the dataset is composed of sequences generated by an Illumina technology, such as MiSeq, with an average read length of 300 bp or greater.

Ray Meta

Create a slurm batch file called ray_batch.sh with the following content:
#! /bin/sh
#SBATCH -J assembly
#SBATCH --output ray.log
#SBATCH --error ray.err
#SBATCH --ntasks ***<CPUs>***

#SBATCH --nodes ***<number_of_nodes>***
*project="~/projects/****<project_code>****"*
results="${project}/results"
assembly="${results}/assembly"
qc="${results}/qc"
*sample="****<sample_name>****"*

outdir="${assembly}/${sample}_assembly";
mkdir ${outdir};
paired="${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz";
singles="{qc}/${sample}.singles.atrim.decontam.qtrim.fq.gz";

ks=(41 51 61 71);
for k in ${ks[@]}; do
* mpirun Ray Meta -k ${k} -minimum-contig-length 200 -i ${paired} -s ${singles} -o ${outdir}/${k};*
done
* the optimal k-mer size to use for de Bruijn graph-based assembly can be difficult to determine. It will depend on various factors, including research goals and the size of the reads generated by the sequencer. As a general rule of thumb, larger values of k tend to provide more accurate assemblies (increased specificity), but at the cost of fewer k-mers being detected (decreased sensitivity). So if genome reconstruction of high abundance community members is the goal, then a larger k-mer size might be preferable. But a smaller k-mer size will probably be more suitable if the focus of the project is on obtaining a more accurate representation of the physiological potential of the community as a whole.
* it is also possible to compare different sized k-mer assemblies to see which size results in the “best” assembly. A batch script can be created, as shown above, that will automate assembly with a range of k-mer sizes. Note that the range of k-mers will be constrained by the length threshold used in the preprocessing steps - the maximum should not exceed the size of the smallest read in the dataset.
* Ray allows for multiple sets of paired-end and single-end reads to be provided together. These data will then get assembled as a consolidated dataset. This can be desirable if multiple libraries with different insert sizes were constructed for a sample, or if multiple samples should be assembled together. To provide multiple datasets to Ray, add additional -p and -s arguments to the command (e.g. `Ray Meta -i <insert_one_pairs> -i <insert_two_pairs> -s <insert_one_singles> -s <insert_two_singles>`).

Assemble paired-end and single-end reads together:
`sbatch ray_batch.sh`

IDBA-UD

Decompress reads file and format as FASTA:

```
srun fastq-to-fasta.py --output ${qc}/${sample}.interleaved.atrim.decontam.qtrim.fa
${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz 2>/dev/null &
```

Assemble paired-end and single-end reads separately:

```
srun --cpus-per-task ${cpus} idba_ud --pre_correction --mink <kmer_minsize> --maxk
<kmer_maxsize> --step <step_size> --min_contig 200 --num_threads ${cpus} -r
${qc}/${sample}.interleaved.atrim.decontam.qtrim.fa -o ${assembly}/${sample}_assembly
&

srun --cpus-per-task ${cpus} idba_ud --pre_correction --mink <kmer_minsize> --maxk
<kmer_maxsize> --step <step_size> --min_contig 200 --num_threads ${cpus} -r
${qc}/${sample}.singles.atrim.decontam.qtrim.fa -o
${assembly}/${sample}_assembly_singles &

rm ${qc}/${sample}.interleaved.atrim.decontam.qtrim.fa
```

* IDBA-UD is an iterative de Bruijn graph assembler. The algorithm will perform successive assemblies starting with a k-mer size of **mink** and ending with **maxk**. After the initial round of assembly, contigs generated from the previous iteration are used to create the de Bruijn graph for the next iteration.
* the final round of assembly will produce a file called contig.fa. This assembly should be used in downstream applications.

MEGAHIT

Assemble paired-end and single-end reads together:

```
srun --cpus-per-task ${cpus} megahit -t ${cpus} --12
${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz --read
${qc}/${sample}.singles.atrim.decontam.qtrim.fq.gz --out-prefix ${sample} --out-dir
${assembly}/${sample}_assembly --min-contig-len 200 --k-min <kmer_minsize_odd> --k-max
<kmer_maxsize_odd> --k-step <step_size_even> &

find ./ -type d -regex ".*intermediate_contigs" -exec rm -R {} \;
```

* like IDBA-UD, MEGAHIT is an iterative de Bruijn graph assembler. The algorithm will perform successive assemblies starting with a k-mer size of **k-min** and ending with **k-max**.
* for high complexity datasets, such as those from soils, the developers of MEGAHIT recommend a high initial value for k (e.g. 27), while a small **k-min** (e.g. 11) is recommended for other low-coverage datasets.
* MEGAHIT allows for multiple sets of paired-end and single-end reads to be provided together. These data will then get assembled as a consolidated dataset. This can be desirable if multiple libraries with different insert sizes were constructed for a sample, or if multiple samples should be assembled together. Multiple datasets can be provided to MEGAHIT by separating the input files by a comma with no spaces (e.g. `megahit --12 <insert_one_pairs>,<insert_two_pairs> --read <insert_one_singles>,<insert_two_singles>`).

MetaSPAdes

Assemble paired-end and single-end reads together:

```
srun --cpus-per-task ${cpus} spades.py --meta -t ${cpus} -m <max_memory_Gb>
--phred-offset 33 -k auto --12 ${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz
-s ${qc}/${sample}.singles.atrim.decontam.qtrim.fq.gz -o
${assembly}/${sample}_assembly &
```

* like IDBA-UD, (Meta)SPAdes is an iterative de Bruijn graph assembler. The algorithm will perform successive assemblies starting from small **k** to large **k**. By default (Meta)SPAdes will automatically determine the appropriate range of k-mers to iterate over based on read length and dataset type, but will accept a custom range of k-mers if a comma-separated list of sizes is provided to the **k** argument.
* the maximum amount of memory to allow MetaSPAdes to use should be a fraction of the total memory of the node the job will run on (e.g. 0.8 * 254 for node0 or node1).
* (Meta)SPAdes allows for multiple sets of paired-end and single-end reads to be provided together. These data will then get assembled as a consolidated dataset. This can be desirable if multiple libraries with different insert sizes were constructed for a sample, or if multiple samples should be assembled together. Multiple datasets can be provided to (Meta)SPAdes by adding additional --pe<#>-12 and --s<#> arguments, where <#> is the library number from 1 to 9 (e.g. `spades.py --pe1-12 <insert_one_pairs> --pe2-12 <insert_two_pairs> --s1 <insert_one_singles> --s2 <insert_two_singles>`). To add more than nine, a YAML dataset file will need to be created and provided to the `--dataset` argument of `spades.py`. See the SPAdes manual for formatting requirements.

DISCO

Assemble paired-end and single-end reads together:

```
runDisco-MPI-SLURM.sh -np <number_nodes> -n ${cpus} -inP
${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz -inS
${qc}/${sample}.singles.atrim.decontam.qtrim.fq.gz -d ${assembly}/${sample}_assembly
-o ${sample}
```

* certain assembly parameters can be configured by providing DISCO with custom parameter files using the -p, -p2, and -p3 arguments. See /usr/local/src/Disco-1.1/disco.cfg for the default assembly configuration.

Final steps

Rename contigs for downstream analysis:

```
srun anvi-script-reformat-fasta ${assembly}/${sample}_assembly/ -o
${assembly}/${sample}_assembly/${sample}.contigs.renamed.fa -l 0 --simplify-names
--report-file ${assembly}/${sample}_assembly/${sample}.contig_names_map.tsv &

find ./ -type f -regex ".*<original contigs filename>" -exec rm {} \;
```

Read Remapping

(optional, if applicable) Demultiplex combined samples:

```
srun demultiplex_by_header --interleaved --format fastq --gzip --suffix
interleaved.atrim.decontam.qtrim.fq --barcodes <barcodes_file> --distance
<hamming_distance_threshold> ${qc}/${combined}.interleaved.atrim.decontam.qtrim.fq.gz
&
```

* this step should only be done if datasets were combined prior to preprocessing.
* headers must be formatted as Casava 1.8 (i.e '@seqid <strand>:N:0:<barcode>').
* the barcodes file contains sample names mapped to the barcode component of the sequence headers, in tabular format. See `demultiplex_by_header --help` for formatting details.
* the hamming distance threshold specifies the maximum distance allowed between the sequence barcode and a template barcode from the barcodes file to be sorted into the same partition. This should normally not be greater than 2.

Index assembly for read re-mapping:

```
srun bowtie2-build ${assembly}/${sample}_assembly/${sample}.contigs.renamed.fa
${assembly}/${sample}_assembly/${sample}.contigs &
```

Calculate the insert size distribution:

```
ishist="${stats}/${sample}.insert_size.hist"

srun bbmap.sh -Xmx24g nodisk interleaved=t
in=${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz out=/dev/null
ihist=${ishist} reads=100000
ref=${assembly}/${sample}_assembly/${sample}.contigs.renamed.fa &
```

Initialize mapping parameters:

```
ismean=$(grep "#Mean" ${ishist} | sed -n "/#Mean\t/s/#Mean\t//p" | xargs printf
'%.*f\n' 0)
issd=$(grep "#STDev" ${ishist} | sed -n "/#STDev\t/s/#STDev\t//p" | xargs printf
'%.*f\n' 0)
ismin=$(value=$(expr ${ismean} - ${issd} \* 3); if [[ ${value} -lt 0 ]]; then echo 0;
else echo $value; fi)
ismax=$(expr ${ismean} + ${issd} \* 3)
```

* accurate short read re-mapping requires inputs for both minimum and maximum insert size. If these values are unknown, the mean and standard deviation can be used to estimate them. One strategy would be to subtract 3 standard deviations from the mean for the minimum and add 3 standard deviations to the mean for the maximum, as shown above. The mean and standard deviation statistics can be found in the insert size histogram generated from the previous step.

Like in preprocessing of the sequencing data, read remapping can be done sequentially or all at once.

Step by step

Map short reads to the assembly:

```
srun --cpus-per-task $(t=(expr ${cpus} - 1); if [[ t -lt 1 ]]; then echo 1; else echo $t; fi) bowtie2 -p $(t=(expr ${cpus} - 1); if [[ t -lt 1 ]]; then echo 1; else echo $t; fi) --very-sensitive -I ${ismin} -X ${ismax} -x ${assembly}/${sample}_assembly/${sample}.contigs --interleaved ${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz 2>${logs}/${sample}.mapping.log | srun samtools -l 9 -T ${sample} -o ${assembly}/${sample}_assembly/${sample}.mapped.sorted.bam &
```

* multiple sets of read pairs can be provided to bowtie2 by separating the file names by a comma with no spaces (e.g. `srun bowtie2 --interleaved <large_insert_pairs>,<small_insert_pairs>`).

Filter mapped reads to only include pairs that aligned somewhere in the assembly:

```
srun samtools view -b -h -F 4 -o ${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.bam ${assembly}/${sample}_assembly/${sample}.mapped.sorted.bam &
```

* an alternative is to use the flag -f 2, which filters mapped reads such that only those that aligned concordantly are kept. An alignment is concordant when both reads in a pair map to the assembly in their correct orientation and within the insert size range.

Dereplicate mapped reads:

```
srun picard -Xmx30g MarkDuplicates COMPRESSION_LEVEL=9 USE_JDK_INFLATER=true USE_JDK_DEFLATER=true REMOVE_DUPLICATES=true INPUT=${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.bam OUTPUT=${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.derep.bam METRICS_FILE=${stats}/${sample}.replicates.txt 2>${logs}/${sample}.derep.log &
```

* dereplication is performed after mapping short reads to an assembly to prevent artificial replicates from impacting coverage estimates. But be aware that it is impossible to discriminate between biological and artificial replicates, so dereplication will consequently remove biological replicates as well if any are present in the data. Discarding these replicates could negatively impact diversity and gene abundance estimates, especially as they are not expected to occur at the same frequency for all members of a community. However, the probability of more than one sequence starting at the same position in metagenome at random is extremely low (Gomez-Alvarez *et al.*, 2009), so two or more reads that do occur at the same position are most likely an artifact of the sequencing process and the safe bet would be to discard them.

All at once

Map short reads to the assembly, dereplicate mapped reads, and filter mapped reads to only include pairs that aligned somewhere in the assembly:

```
srun --cpus-per-task $(t=(expr ${cpus} - 3); if [[ t -lt 1 ]]; then echo 1; else echo $t; fi) bowtie2 -p $(t=(expr ${cpus} - 3); if [[ t -lt 1 ]]; then echo 1; else echo $t; fi) --very-sensitive -I ${ismin} -X ${ismax} -x ${assembly}/${sample}_assembly/${sample}.contigs --interleaved ${qc}/${sample}.interleaved.atrim.decontam.qtrim.fq.gz 2>${logs}/${sample}.mapping.log | srun samtools sort -@ 1 -m 10G -l 0 -O bam -T ${sample} - | srun samtools view -b -u -h -F 4 - | srun picard -Xmx30g MarkDuplicates COMPRESSION_LEVEL=9 USE_JDK_INFLATER=true USE_JDK_DEFLATER=true REMOVE_DUPLICATES=true INPUT=/dev/stdin OUTPUT=${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.derep.bam METRICS_FILE=${stats}/${sample}.replicates.txt 2>${logs}/${sample}.derep.log &
```

Final steps

Index the mapped reads:

```
srun samtools index ${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.derep.bam &
```

Assembly Validation

Obtain assembly statistics:

```
srun --cpus-per-task <number_assemblies_compared> metaquast.py --threads <number_assemblies_compared> --gene-finding --no-check --no-plots --no-icarus --no-snps --max-ref-number 0 --labels "<assembly1_label>,<assembly2_label>,..." --output-dir assembly_stats <assembly1> <assembly2> … &
```

* the resulting output files contain numerous assembly statistics, including number of contigs, largest contig, total size (in base pairs), N50 score, and number of predicted genes. N50 score (or variants NG50 and NGA50) is one of the most commonly used statistics for evaluating an assembly; it is defined as the size of the smallest contig (in bp) such that 50% of the total length of the assembly is made up of contigs that size or larger. The general trend to look for is a high N50 score; however, a high N50 does not necessarily indicate a good assembly. Misassembled contigs - which can occur as a result of the presence of large repetitive regions, high strain variation, or from sequencing errors that cause false alignments - will inflate N50 scores. As much effort as is reasonable should be put into identifying and breaking up misassemblies prior to further analysis.

(optional, if applicable) Compare multiple assemblies of a single sample using an assembly likelihood estimator:

```
srun ALE --qOff 33 --metagenome ${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.derep.bam
```

```
${assembly}/${sample}_assembly/${sample}.contigs.renamed.fa ${stats}/${sample}.ale
>/dev/null 2>${logs}/${sample}.ale.log &

head -n 15 ${stats}/${sample}.ale
```
* the types of errors that ALE is able to detect include chimeras, indels, single-base substitutions, and copy number errors.
* a lower score (higher negative score) means an assembly with more potential errors. In other words, the assembly with ALE score closest to zero should be considered the "best" assembly (best according the the metrics that ALE measures).

(optional, if applicable) Compare multiple assemblies of a single sample using the rate that at which short reads map to the assembly:
```
grep "aligned concordantly exactly 1 time" ${logs}/${sample}.mapping.log
grep "overall alignment rate" ${logs}/${sample}.mapping.log
```
* it is not uncommon for only a small percentage (< 30%) of short reads to successfully map to an assembly, particularly for highly complex communities.

If more than one assembly was generated from a single sample, select the best one to use in downstream analyses based on the above metrics of assembly quality (mapping rate, ALE score, and assembly statistics).

Check if there are any positions with zero coverage for the chosen assembly:
```
srun bedtools genomecov -pc -bg -ibam
${assembly}/${sample}_assembly/${sample}.mapped.sorted.filt.derep.bam | awk '$4 < 1'
```
* the output of this command will be the locations in the assembly that are not covered by at least one fragment. If the program completes without writing anything to the screen, then the assembly has no locations with zero coverage.

Break up contigs at positions of zero coverage. This can be done manually (i.e. via a text editor) using the location information from the previous step.

(optional, if applicable) Obtain the reference coverage of an assembly:
```
refs=($(find /path/to/reference_base_dir -type f -name "*\.fa" ! -name "\.*\.fa"))
assems=($(find ${assembly}/${sample}_assembly/ -maxdepth 2 -type f -regex
".*\.renamed\.fa"))

labels=()
for assem in ${assems[@]}; do
  IFS=". " read -a array <<< $(basename $assem);
  label=${array[0]};
  labels+=("$label");
done
```

```
srun --ntasks ${cpus} metaquast.py --threads ${cpus} --gene-finding --max-ref-number 0
--no-check --output-dir reference_stats -R $(IFS=", "; shift; echo "${refs[*]}")
--labels "$(IFS=", "; shift; echo "${labels[*]}")" ${assems[@]} &
```
* requires reference genomes to be available, either in separate FASTA files or together.

(optional, if applicable) Find the fraction of assembly content that is contained in the reference genomes:

```
for assem in ${assems[@]}; do
  srun sourmash compute --dna --ksizes 21,31,51 --num-hashes 500 --scaled 10000
--output ${assem}.sig ${assem} &  # computes assembly minhash signatures
done

for ref in ${refs[@]}; do
  srun sourmash compute --dna --ksizes 21,31,51 --num-hashes 500 --scaled 10000
--track-abundance --name-from-first --output ${ref}.sig ${ref} &  # computes reference
minhash signatures
done

ksizes=(21 31 51)
for k in ${ksizes[@]}; do
  srun sourmash sbt_index --dna --ksize ${k} /path/to/reference_fastas/references_${k}
/path/to/reference_fastas/*.fa.sig;  # loads sketches into a bloom tree
done

assem_sigs=$(find ./ -type f -name ".*\.contigs\.fa\.sig")
for sig in ${assem_sigs[@]}; do
  for k in ${ksizes}; do
    srun sourmash sbt_gather --dna --ksize ${k}
/path/to/reference_fastas/references_${k}.sbt.json ${sig} 2>
${logs}/${sample}.contigs_ref_match.${k}.log;
  done
done
```
* requires reference genomes to be available, either in separate FASTA files or together.