MA5112

Group 1 Nextflow Script: Hi-C Assay

Gavin Farrell

Sarah Drury

Aodán Laighneach

Eric Tornabell

groupnextflow / hi-c.nf in master

Cancel changes

Edit file | Preview changes

Tabs 8 No wrap

```
#!/bin/bash/env nextflow

/*
 * Written by Sarah
 */

/*
   Perform Hi-C analysis on raw fastq.gz files. Adapted from https://github.com/nf-core/hic/
   Run script using "nextflow run hi-c.nf" Optionally, add "--restriction <sequence>" to
   specify the restriction enzyme cut site. Default is 'A^AGCTT' (digestion by HindIII),
   which was the protocol run on the test dataset provided.
*/

// SETUP STEPS

// Declare pipeline parameters
reads = "$baseDir/data/*_R{1,2}.fastq.gz"
reference = "$baseDir/reference/*.fsa"
scripts = "$baseDir/scripts"
outDir = "$baseDir/results"
project_dir = projectDir

/*
  Default restriction cutting site. Can be overridden by
  using the --'<sequence>' command when launching the script
*/

params.restriction = 'A^AGCTT'


// Set up channels for raw (.fastq.gz) and reference genome (.fsa) files
Channel
        .fromPath(reads)
        .map{file -> [file.simpleName, file]}
        .into{ reads_ch; reads_for_fastqc }

Channel
        .fromPath(reference)
        .map{file -> [file.simpleName, file]}
        .into{ reference_ch; fasta_for_resfrag_ch; fasta_for_chromsize }


// STEP 1: Align reads to genome

// Step by Bowtie2 to index the reference genome
process create_bowtie2_index {
        publishDir path: "$outDir/index", mode: "copy"

        input:
        tuple val(base), file(ref) from reference_ch

        output:
        tuple val(base), file("*.bt2") into index_ch

        script:
        """
        bowtie2-build ${ref} ${base}
        """
}

/*
  Process cuts the reference genome at the restriction enzyme cutting site. The digested reference genome is
  used in the get_valid_interaction process to remove experimental artefacts.
*/
process get_restriction_fragments {
        publishDir path: "$outDir/index", mode: "copy"

```

```
        input:
        file(fasta) from fasta_for_resfrag_ch

        output:
        file("*.bed") into res_frag_ch

        script:
        """
        python $scripts/digest_genome.py \\
                -r ${params.restriction} \\
                -o restriction_fragments.bed ${fasta[1]}
        """
}


// R1 and R2 reads are aligned seperately to the reference genome.
process bowtie2_end_to_end {
        publishDir path: "$outDir/align", mode: "copy"

        input:
        each(reads) from reads_ch
        tuple val(base), file(index) from index_ch

        output:
        tuple val(prefix), file("${name}.bam") into end_to_end_bam

        script:
        name = reads[0]
         prefix = name.toString() - ~/(_R1|_R2|_val_1|_val_2|_1|_2)$/

         """
         bowtie2 -x ${base} \\
                 -U ${reads[1]} \\
                 -S ${reads[0]}.bam \\
                 --very-sensitive \\
                 -L 30 \\
                 --score-min L,-0.6,-0.2 \\
                 --end-to-end \\
                 --reorder
        """
}

// R1 and R2 reads are combined to produce a paired-end bam.
process combine_mapped_files{
        publishDir path: "$outDir/align", mode: "copy",
                saveAs: {filename ->
                filename.indexOf(".pairstat") > 0 ? "$outDir/stats/$filename" : "$filename"}

        input:
        tuple val(sample), file(aligned_bam) from end_to_end_bam.groupTuple()

        output:
        tuple val(sample), file("${sample}_bwt2pairs.bam") into paired_bam
        tuple val(oname), file("*.pairstat") into all_pairstat

        script:
        oname = sample.toString() - ~/(\.[0-9]+)$/

        """
        python $scripts/mergeSAM.py \\
                -f ${aligned_bam[0]} \\
                -r ${aligned_bam[1]} \\
```

```
                -r ${aligned_bam[1]} \\
                -o ${sample}_bwt2pairs.bam \\
                --single --multi -t
        """
}

/*
 * Written by Gavin
 */

// STEP 2: Detection of valid interaction products

// Sets up and defines the  process for getting valid interaction (GVI).
 process get_valid_interaction{
        publishDir path: "$outDir/valid", mode: "copy",
                saveAs: {filename ->
                filename.indexOf(".RSstat") > 0 ? "$outDir/stats/$filename" : "$filename"}

// Defines the input values for the GVI process
        input:
        tuple val(sample), file(pe_bam) from paired_bam
        file(frag_file) from res_frag_ch.collect()

// Defines the output values for the GVI process
        output:
        tuple val(sample), file("*.validPairs") into valid_pairs
        tuple val(sample), file("*.validPairs") into valid_pairs_4cool
        tuple val(sample), file("*.RSstat") into all_rsstat

// Defines the python script for the GVI  process to utilise
        script:
        """
        python $scripts/mapped_2hic_fragments.py \\
                -f ${frag_file} \\
                -r ${pe_bam} \\
                sort -T /tmp/ -k2,2V -k3,3n -k5,5V -k6,6n
        """
}

// STEP 3: Duplicates removal

// Sets up and defines the  process for removing duplicate (RD) reads in the data
process remove_duplicates {
        publishDir path: "$outDir/valid", mode: "copy",
                saveAs: {filename ->
                filename.indexOf(".mergestat") > 0 ? "$outDir/stats/$filename" : "$filename"}

// Defines the input values for the RD process
        input:
        tuple val(sample), file(vpairs) from valid_pairs.groupTuple()

// Defines the output values for the RD process
        output:
        tuple val(sample), file("*.allValidPairs") into all_valid_pairs
        tuple val(sample), file("*.allValidPairs") into all_valid_pairs_4cool
        file("*") into all_mergestat

// Defines the bash script code used for the RD process to operate
        script:
        """
        ## Sort valid pairs and remove read pairs with same starts (i.e duplicated read pairs)
        sort -T /tmp/ -S 50% -k2,2V -k3,3n -k5,5V -k6,6n -m ${vpairs} | \
                awk -F"\\t" 'BEGIN{c1=0;c2=0;s1=0;s2=0}(c1!=\$2 || c2!=\$5 || s1!=\$3 || s2!=\$6){print;c1=\$2;c2=\$5;s1=\$3;s2=\$6}' >> ${sample}.allValidPairs
        echo -n "valid_interaction\t" >> ${sample}_allValidPairs.mergestat
        cat ${vpairs} | wc -l >> ${sample}_allValidPairs.mergestat
        echo -n "valid_interaction_rmdup\t" >> ${sample}_allValidPairs.mergestat
        cat ${sample}.allValidPairs | wc -l >> ${sample}_allValidPairs.mergestat
```

```
        ## Count short range (<20000) vs long range contacts
        awk 'BEGIN{cis=0;trans=0;sr=0;lr=0} \$2 == \$5{cis=cis+1; d=\$6>\$3?\$6-\$3:\$3-\$6; if (d<=20000){sr=sr+1}else{lr=lr+1}} \$2!=\$5{trans=trans+1}END{print "trans_interact
        """
}


/*
 * Written by Aodán
 */

// STEP 4: Generate raw and normalized contact maps

 /*
 *Chromsome/scaffold sizes must be provided to build contact maps. First, samtools is used
 *to index reference fasta to .fai file. Columns 1 & 2 (containing chromosome/scaffold) I.D. and
 *length respectively will be cut to chrom.size file for future use.
*/

process make_chrom_size {
        publishDir path: "$outDir/chrom_size", mode: "copy"

        input:
        tuple val(base), file(fasta) from fasta_for_chromsize

        output:
        file("*.size") into chromosome_size, chromosome_size_cool

        script:
        """
        samtools faidx ${fasta}
        cut -f1,2 ${fasta}.fai > chrom.size
        """
}

 /*
  *External program file build_matrix is used to produce contact map(s). This will require
  *inputs of resolution, chromosome length and file denoting valid interaction pairs. Valid pairs
  *and chromsome lengths will be collected from established channels above. Resolution will be
  *provided through bin_size parameter, where two resolutions will be considered. Output will consist of
  *genomic interval files (.BED) consistent with resolution and raw (unnormalised) matrix file.
 */

 // Resolutions for contact maps
 bin_size = '1000000,500000'
 map_res = bin_size.tokenize(',')

 process build_contact_maps {
         publishDir path: "$outDir/matrix/raw", mode: "copy"

         input:
         tuple val(sample), file(vpairs), val(mres) from all_valid_pairs.combine(map_res)
         file chrsize from chromosome_size.collect()

         output:
         file("*.matrix") into raw_maps
         file "*.bed"

         script:
         """
         ${scripts}/build_matrix --matrix-format upper  \\
                 --binsize ${mres} \\
                 --chrsizes ${chrsize} \\
                 --ifile ${vpairs} \\
                 --oprefix ${sample}_${mres}
         """
 }
```

```
262
263
264 /*
265  *Biases in the raw matrix file will be normalised using ICE. Raw matrix file will be used from
266  *previously-established raw_maps channel. Low and high counts will be prefiltered before matrix is
267  *normalised. Mamimum iterations for normalisation will be set at 100. Output consists of normalised (iced)
268  *matrices for specified resolutions.
269 */
270
271 process run_ice{
272         publishDir "$outDir/matrix/iced", mode: "copy"
273
274         input:
275         file(rmaps) from raw_maps
276
277         output:
278         file("*iced.matrix") into iced_maps
279
280         script:
281         prefix = rmaps.toString() - ~/(\.matrix)?$/
282         """
283         ice --filter_low_counts_perc 0.02 \
284                 --results_filename ${prefix}_iced.matrix \
285                 --filter_high_counts_perc 0 \
286                 --max_iter 100 \
287                 --eps 0.1 \
288                 --remove-all-zeros-loci \
289                 --output-bias 1 \
290                 --verbose 1 ${rmaps}
291         """
292 }
293
294
```

```
295 /*
296  * Written by Eric
297  */
298
299 // STEP 5: Generate statistics files and quality control report
300 /*
301   Combines the R1 and R2 statistics files. Statistics about read pairs filtering are available in the
302   .mRSstat file, and pairing statistics are available in the .mpairstat file.
303 */
304
305 process merge_stats {
306         publishDir "$outDir/mstats", mode: "copy"
307
308         input:
309         tuple val(prefix), file(fstat) from all_rsstat.groupTuple().concat(all_pairstat.groupTuple())
310
311         output:
312         file("*") into all_mstats
313
314         script:
315         sample = prefix.toString() - ~/(_R1|_R2|_val_1|_val_2|_1|_2)/
316         if ( (fstat =~ /.pairstat/) ){ ext = "mpairstat" }
317         if ( (fstat =~ /.RSstat/) ){ ext = "mRSstat" }
318
319         """
320         python $scripts/merge_statfiles.py -f ${fstat} > ${prefix}.${ext}
321         """
322 }
323
324
325 // Perform quality control using FastQC
326 process fastqc {
327         input:
328         tuple val(sample), file(reads) from reads_for_fastqc
329
```

```
329
330         output:
331         file "*_fastqc.{zip,html}" into fastqc_results
332
333         script:
334         """
335         fastqc -q ${reads}
336         """
337 }
338
339 /*
340   Finally MultiQC process is a reporting tool that parses summary statistics from results and log files generated by other
341   bioinformatics tools. It recursively searches through all provided file paths and finds files that it recognizes. It parses
342   relevant information from these and generates a single stand-alone HTML report file.
343 */
344 process multiqc {
345         publishDir "$outDir", mode: "copy"
346
347         input:
348         file ('*') from fastqc_results.collect().ifEmpty([])
349
350         output:
351         file "multiqc_report.html" into multiqc_report
352         file "multiqc_data"
353
354         script:
355         """
356         multiqc .
357         """
358 }
```