systemPipeR: NGS workflow and report generation environment

Author: Daniela Cassol (danielac@ucr.edu) and Thomas Girke (thomas.girke@ucr.edu)

Last update: 02 February, 2019

Package

systemPipeR 1.17.5

Contents

Note: the most recent version of this tutorial can be found here and a short overview slide show here.

Note: if you use `systemPipeR` in published research, please cite:

Backman, T.W.H and Girke, T. (2016). *systemPipeR*: NGS Workflow and Report Generation Environment. *BMC Bioinformatics*, 17: 388. 10.1186/s12859-016-1241-0.

1 Introduction

`systemPipeR` provides utilities for building and running automated end-to-end analysis workflows for a wide range of next generation sequence (NGS) applications such as RNA-Seq, ChIP-Seq, VAR-Seq and Ribo-Seq (Girke 2014). Important features include a uniform workflow interface across different NGS applications, automated report generation, and support for running both R and command-line software, such as NGS aligners or peak/variant callers, on local computers or compute clusters. The latter supports interactive job submissions and batch submissions to queuing systems of clusters. For instance, `systemPipeR` can be used with most command-line aligners such as `BWA` (Heng Li 2013; H Li and Durbin 2009), `HISAT2` (Kim, Langmead, and Salzberg 2015), `TopHat2` (Kim et al. 2013) and `Bowtie2` (Langmead and Salzberg 2012), as well as the R-based NGS aligners `Rsubread` (Liao, Smyth, and Shi 2013) and `gsnap (gmapR)` (Wu and Nacu 2010). Efficient handling of complex sample sets (*e.g.* FASTQ/BAM files) and experimental designs is facilitated by a well-defined sample annotation infrastructure which improves reproducibility and user-friendliness of many typical analysis workflows in the NGS area (Lawrence et al. 2013).

Motivation and advantages of `sytemPipeR` environment:

1. Facilitates design of complex NGS workflows involving multiple R/Bioconductor packages
2. Common workflow interface for different NGS applications
3. Makes NGS analysis with Bioconductor utilities more accessible to new users
4. Simplifies usage of command-line software from within R
5. Reduces complexity of using compute clusters for R and command-line software
6. Accelerates runtime of workflows via parallelzation on computer systems with mutiple CPU cores and/or multiple compute nodes
7. Automates generation of analysis reports to improve reproducibility

A central concept for designing workflows within the `sytemPipeR` environment is the use of workflow management containers called `SYSargs` (see Figure 1). Instances of this S4 object class are constructed by the `systemArgs` function from two simple tabular files: a `targets` file and a `param` file. The latter is optional for workflow steps lacking command-line software. Typically, a `SYSargs` instance stores all sample-level inputs as well as the paths to the corresponding outputs generated by command-line- or R-based software generating sample-level output files, such as read preprocessors (trimmed/filtered FASTQ files), aligners (SAM/BAM files), variant callers (VCF/BCF files) or peak callers (BED/WIG files). Each sample level input/outfile operation uses its own `SYSargs` instance. The outpaths of `SYSargs` usually define the sample inputs for the next `SYSargs` instance. This connectivity is established by writing the outpaths with the `writeTargetsout` function to a new `targets` file that serves as input to the next `systemArgs` call. Typically, the user has to provide only the initial `targets` file. All downstream `targets` files are generated automatically. By chaining several `SYSargs` steps together one can construct complex workflows involving many sample-level input/output file operations with any combinaton of command-line or R-based software.

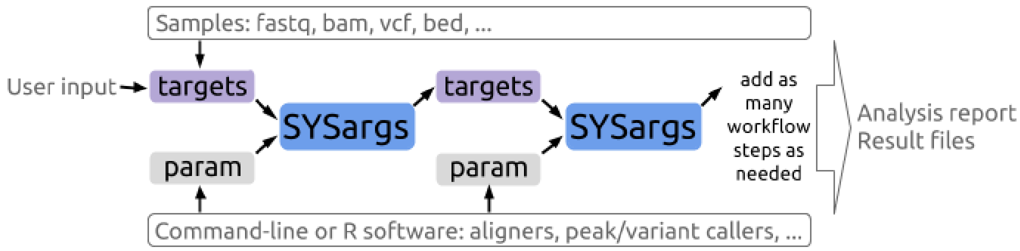

Figure 1: Workflow design structure of *systemPipeR*

The intended way of running `sytemPipeR` workflows is via `*.Rmd` or `*.Rnw` files, which can be executed either line-wise in interactive mode or with a single command from R or the command-line using a `Makefile`. This way comprehensive and reproducible analysis reports can be generated in PDF or HTML format in a fully automated manner by making use of the highly functional reporting utilities available for R. Templates for setting up custom project reports are provided as `*.Rmd` files by the helper package `systemPipeRdata` and in the vignettes subdirectory of `systemPipeR`. The corresponding PDFs of these report templates are available here: `systemPipeRNAseq`, `systemPipeRIBOseq`, `systemPipeChIPseq` and `systemPipeVARseq`. To work with `*.Rnw` or `*.Rmd` files efficiently, basic knowledge of `Sweave` or `knitr` and `Latex` or `R Markdown v2` is required.

2 Getting Started

2.1 Installation

The R software for running `systemPipeR` can be downloaded from *CRAN*. The `systemPipeR` environment can be installed from the R console using the `BiocManager::install` command. The associated data package `systemPipeRdata` can be installed the same way. The latter is a helper package for generating `systemPipeR` workflow environments with a single command containing all parameter files and sample data required to quickly test and run workflows.

```
if (!requireNamespace("BiocManager", quietly = TRUE)) install.packages("BiocManager")
BiocManager::install("systemPipeR")
BiocManager::install("systemPipeRdata")
```

2.2 Loading package and documentation

```
library("systemPipeR")  # Loads the package
library(help = "systemPipeR")  # Lists package info
vignette("systemPipeR")  # Opens vignette
```

2.3 Load sample data and workflow templates

The mini sample FASTQ files used by this overview vignette as well as the associated workflow reporting vignettes can be loaded via the `systemPipeRdata` package as shown below. The chosen data set `SRP010938` contains 18 paired-end (PE) read sets from *Arabidposis thaliana*

(Howard et al. 2013). To minimize processing time during testing, each FASTQ file has been subsetted to 90,000-100,000 randomly sampled PE reads that map to the first 100,000 nucleotides of each chromosome of the *A. thalina* genome. The corresponding reference genome sequence (FASTA) and its GFF annotion files (provided in the same download) have been truncated accordingly. This way the entire test sample data set requires less than 200MB disk storage space. A PE read set has been chosen for this test data set for flexibility, because it can be used for testing both types of analysis routines requiring either SE (single end) reads or PE reads.

The following generates a fully populated `systemPipeR` workflow environment (here for RNA-Seq) in the current working directory of an R session. At this time the package includes workflow templates for RNA-Seq, ChIP-Seq, VAR-Seq and Ribo-Seq. Templates for additional NGS applications will be provided in the future.

```
library(systemPipeRdata)
genWorkenvir(workflow = "rnaseq")
setwd("rnaseq")
```

The working environment of the sample data loaded in the previous step contains the following preconfigured directory structure. Directory names are indicated in ***grey***. Users can change this structure as needed, but need to adjust the code in their workflows accordingly.

- ***workflow/*** (*e.g. rnaseq/*)
  - This is the directory of the R session running the workflow.
  - Run script (**.Rmd* or **.Rnw*) and sample annotation (*targets.txt*) files are located here.
  - Note, this directory can have any name (*e.g.* ***rnaseq***, ***varseq***). Changing its name does not require any modifications in the run script(s).
  - Important subdirectories:
    - ***param/***
      - Stores parameter files such as: **.param*, **.tmpl* and **_run.sh*.
    - ***data/***
      - FASTQ samples
      - Reference FASTA file
      - Annotations
      - etc.
    - ***results/***
      - Alignment, variant and peak files (BAM, VCF, BED)
      - Tabular result files
      - Images and plots
      - etc.

The following parameter files are included in each workflow template:

1. `targets.txt`: initial one provided by user; downstream `targets_*.txt` files are generated automatically
2. `*.param`: defines parameter for input/output file operations, *e.g.* `trim.param`, `bwa.param`, `vartools.parm`, ...
3. `*_run.sh`: optional bash script, *e.g.*: `gatk_run.sh`
4. Compute cluster environment (skip on single machine):
   - `.batchtools.conf.R`: defines type of scheduler for `batchtools`. Note, it is necessary to point the right template accordingly to the cluster in use.
   - `*.tmpl`: specifies parameters of scheduler used by a system, *e.g.* Torque, SGE, Slurm, etc.

2.4 Structure of `targets` file

The `targets` file defines all input files (*e.g.* FASTQ, BAM, BCF) and sample comparisons of an analysis workflow. The following shows the format of a sample `targets` file included in the package. It also can be viewed and downloaded from `systemPipeR`'s GitHub repository here. In a target file with a single type of input files, here FASTQ files of single end (SE) reads, the first three columns are mandatory including their column names, while it is four mandatory columns for FASTQ files of PE reads. All subsequent columns are optional and any number of additional columns can be added as needed.

2.4.1 Structure of `targets` file for single end (SE) samples

```
library(systemPipeR)
targetspath <- system.file("extdata", "targets.txt", package = "systemPipeR")
read.delim(targetspath, comment.char = "#")
##                    FileName SampleName Factor SampleLong
## 1  ./data/SRR446027_1.fastq        M1A     M1  Mock.1h.A
## 2  ./data/SRR446028_1.fastq        M1B     M1  Mock.1h.B
## 3  ./data/SRR446029_1.fastq        A1A     A1   Avr.1h.A
## 4  ./data/SRR446030_1.fastq        A1B     A1   Avr.1h.B
## 5  ./data/SRR446031_1.fastq        V1A     V1   Vir.1h.A
## 6  ./data/SRR446032_1.fastq        V1B     V1   Vir.1h.B
## 7  ./data/SRR446033_1.fastq        M6A     M6  Mock.6h.A
## 8  ./data/SRR446034_1.fastq        M6B     M6  Mock.6h.B
## 9  ./data/SRR446035_1.fastq        A6A     A6   Avr.6h.A
## 10 ./data/SRR446036_1.fastq        A6B     A6   Avr.6h.B
## 11 ./data/SRR446037_1.fastq        V6A     V6   Vir.6h.A
## 12 ./data/SRR446038_1.fastq        V6B     V6   Vir.6h.B
## 13 ./data/SRR446039_1.fastq       M12A    M12 Mock.12h.A
## 14 ./data/SRR446040_1.fastq       M12B    M12 Mock.12h.B
## 15 ./data/SRR446041_1.fastq       A12A    A12  Avr.12h.A
## 16 ./data/SRR446042_1.fastq       A12B    A12  Avr.12h.B
## 17 ./data/SRR446043_1.fastq       V12A    V12  Vir.12h.A
## 18 ./data/SRR446044_1.fastq       V12B    V12  Vir.12h.B
##    Experiment        Date
## 1           1 23-Mar-2012
## 2           1 23-Mar-2012
## 3           1 23-Mar-2012
## 4           1 23-Mar-2012
## 5           1 23-Mar-2012
## 6           1 23-Mar-2012
## 7           1 23-Mar-2012
## 8           1 23-Mar-2012
## 9           1 23-Mar-2012
## 10          1 23-Mar-2012
## 11          1 23-Mar-2012
## 12          1 23-Mar-2012
## 13          1 23-Mar-2012
## 14          1 23-Mar-2012
```

```
## 15          1 23-Mar-2012
## 16          1 23-Mar-2012
## 17          1 23-Mar-2012
## 18          1 23-Mar-2012
```

To work with custom data, users need to generate a `targets` file containing the paths to their own FASTQ files and then provide under `targetspath` the path to the corresponding `targets` file.

2.4.2 Structure of `targets` file for paired end (PE) samples

```
targetspath <- system.file("extdata", "targetsPE.txt", package = "systemPipeR")
read.delim(targetspath, comment.char = "#")[1:2, 1:6]
##                   FileName1                 FileName2
## 1 ./data/SRR446027_1.fastq ./data/SRR446027_2.fastq
## 2 ./data/SRR446028_1.fastq ./data/SRR446028_2.fastq
##   SampleName Factor SampleLong Experiment
## 1        M1A     M1  Mock.1h.A          1
## 2        M1B     M1  Mock.1h.B          1
```

2.4.3 Sample comparisons

Sample comparisons are defined in the header lines of the `targets` file starting with '`# <CMP>`'.

```
readLines(targetspath)[1:4]
## [1] "# Project ID: Arabidopsis - Pseudomonas alternative splicing study (SRA: SRP010938; PMID: 24098335)"
## [2] "# The following line(s) allow to specify the contrasts needed for comparative analyses, such as DEG
## [3] "# <CMP> CMPset1: M1-A1, M1-V1, A1-V1, M6-A6, M6-V6, A6-V6, M12-A12, M12-V12, A12-V12"
## [4] "# <CMP> CMPset2: ALL"
```

The function `readComp` imports the comparison information and stores it in a `list`. Alternatively, `readComp` can obtain the comparison information from the corresponding `SYSargs` object (see below). Note, these header lines are optional. They are mainly useful for controlling comparative analyses according to certain biological expectations, such as identifying differentially expressed genes in RNA-Seq experiments based on simple pair-wise comparisons.

```
readComp(file = targetspath, format = "vector", delim = "-")
## $CMPset1
## [1] "M1-A1"   "M1-V1"   "A1-V1"   "M6-A6"   "M6-V6"
## [6] "A6-V6"   "M12-A12" "M12-V12" "A12-V12"
##
## $CMPset2
##  [1] "M1-A1"   "M1-V1"   "M1-M6"   "M1-A6"   "M1-V6"
##  [6] "M1-M12"  "M1-A12"  "M1-V12"  "A1-V1"   "A1-M6"
## [11] "A1-A6"   "A1-V6"   "A1-M12"  "A1-A12"  "A1-V12"
## [16] "V1-M6"   "V1-A6"   "V1-V6"   "V1-M12"  "V1-A12"
## [21] "V1-V12"  "M6-A6"   "M6-V6"   "M6-M12"  "M6-A12"
## [26] "M6-V12"  "A6-V6"   "A6-M12"  "A6-A12"  "A6-V12"
```

```
## [31] "V6-M12"  "V6-A12"  "V6-V12"  "M12-A12" "M12-V12"
## [36] "A12-V12"
```

2.5 Structure of *param* file and *SYSargs* container

The *param* file defines the parameters of a chosen command-line software. The following shows the format of a sample *param* file provided by this package.

```
parampath <- system.file("extdata", "tophat.param", package = "systemPipeR")
read.delim(parampath, comment.char = "#")
##      PairSet        Name
## 1    modules        <NA>
## 2    modules        <NA>
## 3   software        <NA>
## 4      cores          -p
## 5      other        <NA>
## 6   outfile1          -o
## 7   outfile1        path
## 8   outfile1      remove
## 9   outfile1      append
## 10  outfile1 outextension
## 11 reference        <NA>
## 12   infile1        <NA>
## 13   infile1        path
## 14   infile2        <NA>
## 15   infile2        path
##                                    Value
## 1                           bowtie2/2.2.5
## 2                          tophat/2.0.14
## 3                                 tophat
## 4                                      4
## 5  -g 1 --segment-length 25 -i 30 -I 3000
## 6                             <FileName1>
## 7                              ./results/
## 8                                   <NA>
## 9                                .tophat
## 10             .tophat/accepted_hits.bam
## 11                  ./data/tair10.fasta
## 12                            <FileName1>
## 13                                  <NA>
## 14                            <FileName2>
## 15                                  <NA>
```

The *systemArgs* function imports the definitions of both the *param* file and the *targets* file, and stores all relevant information in a *SYSargs* object (S4 class). To run the pipeline without command-line software, one can assign *NULL* to *sysma* instead of a *param* file. In addition, one can start *systemPipeR* workflows with pre-generated BAM files by providing a targets file where the *FileName* column provides the paths to the BAM files. Note, in the following example the usage of *suppressWarnings()* is only relevant for building this vignette. In typical workflows it should be removed.

```
args <- suppressWarnings(systemArgs(sysma = parampath, mytargets = targetspath))
args
## An instance of 'SYSargs' for running 'tophat' on 18 samples
```

Several accessor methods are available that are named after the slot names of the `SYSargs` object.

```
names(args)
##  [1] "targetsin"     "targetsout"    "targetsheader"
##  [4] "modules"       "software"      "cores"
##  [7] "other"         "reference"     "results"
## [10] "infile1"       "infile2"       "outfile1"
## [13] "sysargs"       "outpaths"
```

Of particular interest is the `sysargs()` method. It constructs the system commands for running command-lined software as specified by a given `param` file combined with the paths to the input samples (*e.g.* FASTQ files) provided by a `targets` file. The example below shows the `sysargs()` output for running TopHat2 on the first PE read sample. Evaluating the output of `sysargs()` can be very helpful for designing and debugging `param` files of new command-line software or changing the parameter settings of existing ones.

```
sysargs(args)[1]
##
## "tophat -p 4 -g 1 --segment-length 25 -i 30 -I 3000 -o /home/dcassol/danielac@ucr.edu/github/systemPipeR/_
modules(args)
## [1] "bowtie2/2.2.5" "tophat/2.0.14"
cores(args)
## [1] 4
outpaths(args)[1]
##
## "/home/dcassol/danielac@ucr.edu/github/systemPipeR/_vignettes/10_Rworkflows/results/SRR446027_1.fastq.toph
```

The content of the `param` file can also be returned as JSON object as follows (requires `rjson` package).

```
systemArgs(sysma = parampath, mytargets = targetspath, type = "json")
## [1] "{\"modules\":{\"n1\":\"\",\"v2\":\"bowtie2/2.2.5\",\"n1\":\"\",\"v2\":\"tophat/2.0.14\"},\"software\"
```

3 Workflow overview

3.1 Define environment settings and samples

A typical workflow starts with generating the expected working environment containing the proper directory structure, input files and parameter settings. To simplify this task, one can load one of the existing NGS workflows templates provided by `systemPipeRdata` into the current working directory. The following does this for the `rnaseq` template. The name of the resulting workflow directory can be specified under the `mydirname` argument. The default `NULL` uses the name of the chosen workflow. An error is issued if a directory of the same name and path exists already. On Linux and OS X systems one can also create new workflow instances from the command-line of a terminal as shown here. To apply workflows to custom data, the user needs to modify the *targets* file and if necessary update the corresponding *param* file(s). A collection of pre-generated *param* files is provided in the *param* subdirectory of each workflow template. They are also viewable in the GitHub repository of *systemPipeRdata* (see here).

```
library(systemPipeR)
library(systemPipeRdata)
genWorkenvir(workflow = "rnaseq", mydirname = NULL)
setwd("rnaseq")
```

Construct *SYSargs* object from *param* and *targets* files.

```
args <- systemArgs(sysma = "param/trim.param", mytargets = "targets.txt")
```

3.2 Read Preprocessing

The function *preprocessReads* allows to apply predefined or custom read preprocessing functions to all FASTQ files referenced in a *SYSargs* container, such as quality filtering or adaptor trimming routines. The paths to the resulting output FASTQ files are stored in the *outpaths* slot of the *SYSargs* object. Internally, *preprocessReads* uses the *FastqStreamer* function from the *ShortRead* package to stream through large FASTQ files in a memory-efficient manner. The following example performs adaptor trimming with the *trimLRPatterns* function from the *Biostrings* package. After the trimming step a new targets file is generated (here *targets_trim.txt*) containing the paths to the trimmed FASTQ files. The new targets file can be used for the next workflow step with an updated *SYSargs* instance, *e.g.* running the NGS alignments with the trimmed FASTQ files.

```
preprocessReads(args = args, Fct = "trimLRPatterns(Rpattern='GCCCGGGTAA',
                subject=fq)",
    batchsize = 1e+05, overwrite = TRUE, compress = TRUE)
writeTargetsout(x = args, file = "targets_trim.txt")
```

The following example shows how one can design a custom read preprocessing function using utilities provided by the *ShortRead* package, and then run it in batch mode with the *'preprocessReads'* function (here on paired-end reads).

```
args <- systemArgs(sysma = "param/trimPE.param", mytargets = "targetsPE.txt")
filterFct <- function(fq, cutoff = 20, Nexceptions = 0) {
    qcount <- rowSums(as(quality(fq), "matrix") <= cutoff, na.rm = TRUE)
    # Retains reads where Phred scores are >= cutoff with N
    # exceptions
    fq[qcount <= Nexceptions]
}
preprocessReads(args = args, Fct = "filterFct(fq, cutoff=20, Nexceptions=0)",
    batchsize = 1e+05)
writeTargetsout(x = args, file = "targets_PEtrim.txt")
```

3.3 FASTQ quality report

The following *seeFastq* and *seeFastqPlot* functions generate and plot a series of useful quality statistics for a set of FASTQ files including per cycle quality box plots, base proportions, base-level quality trends, relative k-mer diversity, length and occurrence distribution of reads, number of reads above quality cutoffs and mean quality distribution.
The function *seeFastq* computes the quality statistics and stores the results in a relatively small list object that can be saved to disk with *save()* and reloaded with *load()* for later plotting. The argument *klength* specifies the k-mer length and *batchsize* the number of reads to random sample from each FASTQ file.

```
fqlist <- seeFastq(fastq = infile1(args), batchsize = 10000,
    klength = 8)
pdf("./results/fastqReport.pdf", height = 18, width = 4 * length(fqlist))
seeFastqPlot(fqlist)
dev.off()
```

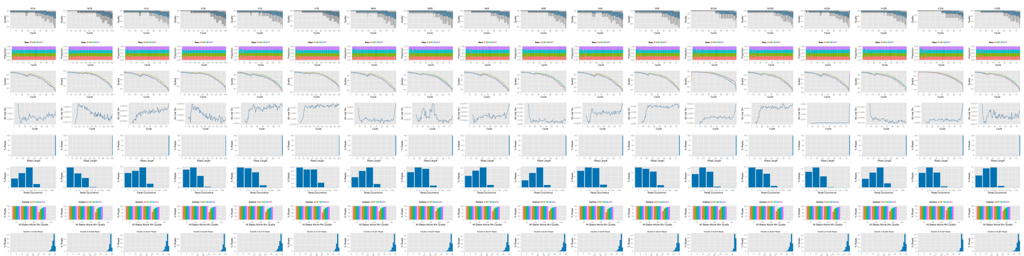

Figure 2: FASTQ quality report

Parallelization of QC report on single machine with multiple cores

```
args <- systemArgs(sysma = "param/tophat.param", mytargets = "targets.txt")
f <- function(x) seeFastq(fastq = infile1(args)[x], batchsize = 1e+05,
    klength = 8)
fqlist <- bplapply(seq(along = args), f, BPPARAM = MulticoreParam(workers = 8))
seeFastqPlot(unlist(fqlist, recursive = FALSE))
```

Parallelization of QC report via scheduler (*e.g.* Slurm) across several compute nodes

```
library(BiocParallel)
library(batchtools)
f <- function(x) {
    library(systemPipeR)
    args <- systemArgs(sysma = "param/tophat.param", mytargets = "targets.txt")
    seeFastq(fastq = infile1(args)[x], batchsize = 1e+05, klength = 8)
}
resources <- list(walltime = 120, ntasks = 1, ncpus = cores(args),
    memory = 1024)
param <- BatchtoolsParam(workers = 4, cluster = "slurm", template = "batchtools.slurm.tmpl",
    resources = resources)
fqlist <- bplapply(seq(along = args), f, BPPARAM = param)
seeFastqPlot(unlist(fqlist, recursive = FALSE))
```

3.4 Alignment with `Tophat2`

Build `Bowtie2` index.

```
args <- systemArgs(sysma = "param/tophat.param", mytargets = "targets.txt")
moduleload(modules(args))  # Skip if module system is not available
system("bowtie2-build ./data/tair10.fasta ./data/tair10.fasta")
```

Execute `SYSargs` on a single machine without submitting to a queuing system of a compute cluster. This way the input FASTQ files will be processed sequentially. If available, multiple CPU cores can be used for processing each file. The number of CPU cores (here 4) to use for each process is defined in the `*.param` file. With `cores(args)` one can return this value from the `SYSargs` object. Note, if a module system is not installed or used, then the corresponding `*.param` file needs to be edited accordingly by either providing an empty field in the line(s) starting with `module` or by deleting these lines.

```
bampaths <- runCommandline(args = args)
```

Alternatively, the computation can be greatly accelerated by processing many files in parallel using several compute nodes of a cluster, where a scheduling/queuing system is used for load balancing. To avoid over-subscription of CPU cores on the compute nodes, the value from `cores(args)` is passed on to the submission command, here `nodes` in the `resources` list object. The number of independent parallel cluster processes is defined under the `Njobs` argument. The following example will run 18 processes in parallel using for each 4 CPU cores. If the resources available on a cluster allow to run all 18 processes at the same time then the shown sample submission will utilize in total 72 CPU cores. Note, `clusterRun` can be used with most queueing systems as it is based on utilities from the `batchtools` package which supports the use of template files (`*.tmpl`) for defining the run parameters of different schedulers. To run the following code, one needs to have both a conf file (see `.batchtools.conf.R` samples here) and a template file (see `*.tmpl` samples here) for the queueing available on a system. The following example uses the sample conf and template files for the Slurm scheduler provided by this package.

```
resources <- list(walltime = 120, ntasks = 1, ncpus = cores(args),
    memory = 1024)
reg <- clusterRun(args, conffile = ".batchtools.conf.R", Njobs = 18,
    template = "batchtools.slurm.tmpl", runid = "01", resourceList = resources)
waitForJobs(reg = reg)
```

Useful commands for monitoring progress of submitted jobs

```
getStatus(reg = reg)
file.exists(outpaths(args))
sapply(1:length(args), function(x) loadResult(reg, id = x))
# Works after job completion
```

3.5 Read and alignment count stats

Generate table of read and alignment counts for all samples.

```
read_statsDF <- alignStats(args)
write.table(read_statsDF, "results/alignStats.xls", row.names = FALSE,
    quote = FALSE, sep = "\t")
```

The following shows the first four lines of the sample alignment stats file provided by the *systemPipeR* package. For simplicity the number of PE reads is multiplied here by 2 to approximate proper alignment frequencies where each read in a pair is counted.

```
read.table(system.file("extdata", "alignStats.xls", package = "systemPipeR"),
    header = TRUE)[1:4, ]
##   FileName Nreads2x Nalign Perc_Aligned Nalign_Primary
## 1      M1A   192918 177961     92.24697         177961
## 2      M1B   197484 159378     80.70426         159378
## 3      A1A   189870 176055     92.72397         176055
## 4      A1B   188854 147768     78.24457         147768
##   Perc_Aligned_Primary
## 1             92.24697
## 2             80.70426
## 3             92.72397
## 4             78.24457
```

Parallelization of read/alignment stats on single machine with multiple cores.

```
f <- function(x) alignStats(args[x])
read_statsList <- bplapply(seq(along = args), f, BPPARAM = MulticoreParam(workers = 8))
read_statsDF <- do.call("rbind", read_statsList)
```

Parallelization of read/alignment stats via scheduler (*e.g.* Slurm) across several compute nodes.

```
library(BiocParallel)
library(batchtools)
f <- function(x) {
    library(systemPipeR)
    args <- systemArgs(sysma = "param/tophat.param", mytargets = "targets.txt")
    alignStats(args[x])
}
resources <- list(walltime = 120, ntasks = 1, ncpus = cores(args),
    memory = 1024)
param <- BatchtoolsParam(workers = 4, cluster = "slurm", template = "batchtools.slurm.tmpl",
    resources = resources)
read_statsList <- bplapply(seq(along = args), f, BPPARAM = param)
read_statsDF <- do.call("rbind", read_statsList)
```

3.6 Create symbolic links for viewing BAM files in IGV

The genome browser IGV supports reading of indexed/sorted BAM files via web URLs. This way it can be avoided to create unnecessary copies of these large files. To enable this approach, an HTML directory with http access needs to be available in the user account (*e.g.* *home/publichtml*) of a system. If this is not the case then the BAM files need to be moved or copied to the system where IGV runs. In the following, *htmldir* defines the path to the HTML directory with http access where the symbolic links to the BAM files will be stored. The corresponding URLs will be written to a text file specified under the `_urlfile_` argument.

```
symLink2bam(sysargs = args, htmldir = c("~/.html/", "somedir/"),
    urlbase = "http://myserver.edu/~username/", urlfile = "IGVurl.txt")
```

3.7 Alternative NGS Aligners

3.7.1 Alignment with `Bowtie2` (*e.g.* for miRNA profiling)

The following example runs `Bowtie2` as a single process without submitting it to a cluster.

```
args <- systemArgs(sysma = "param/bowtieSE.param", mytargets = "targets.txt")
moduleload(modules(args))  # Skip if module system is not available
bampaths <- runCommandline(args = args)
```

Alternatively, submit the job to compute nodes.

```
resources <- list(walltime = 120, ntasks = 1, ncpus = cores(args),
    memory = 1024)
reg <- clusterRun(args, conffile = ".batchtools.conf.R", Njobs = 18,
    template = "batchtools.slurm.tmpl", runid = "01", resourceList = resources)
waitForJobs(reg = reg)
```

3.7.2 Alignment with `BWA-MEM` (*e.g.* for VAR-Seq)

The following example runs BWA-MEM as a single process without submitting it to a cluster.

```
args <- systemArgs(sysma = "param/bwa.param", mytargets = "targets.txt")
moduleload(modules(args))  # Skip if module system is not available
system("bwa index -a bwtsw ./data/tair10.fasta")  # Indexes reference genome
bampaths <- runCommandline(args = args[1:2])
```

3.7.3 Alignment with `Rsubread` (*e.g.* for RNA-Seq)

The following example shows how one can use within the *systemPipeR* environment the R-based aligner *Rsubread* or other R-based functions that read from input files and write to output files.

```
library(Rsubread)
args <- systemArgs(sysma = "param/rsubread.param", mytargets = "targets.txt")
# Build indexed reference genome
buildindex(basename = reference(args), reference = reference(args))
align(index = reference(args), readfile1 = infile1(args), input_format = "FASTQ",
    output_file = outfile1(args), output_format = "SAM", nthreads = 8,
    indels = 1, TH1 = 2)
for (i in seq(along = outfile1(args))) asBam(file = outfile1(args)[i],
```

```
    destination = gsub(".sam", "", outfile1(args)[i]), overwrite = TRUE,
    indexDestination = TRUE)
```

3.7.4 Alignment with `gsnap` (*e.g.* for VAR-Seq and RNA-Seq)

Another R-based short read aligner is `gsnap` from the `gmapR` package (Wu and Nacu 2010). The code sample below introduces how to run this aligner on multiple nodes of a compute cluster.

```
library(gmapR)
library(BiocParallel)
library(batchtools)
args <- systemArgs(sysma = "param/gsnap.param", mytargets = "targetsPE.txt")
gmapGenome <- GmapGenome(reference(args), directory = "data",
    name = "gmap_tair10chr/", create = TRUE)
f <- function(x) {
    library(gmapR)
    library(systemPipeR)
    args <- systemArgs(sysma = "param/gsnap.param", mytargets = "targetsPE.txt")
    gmapGenome <- GmapGenome(reference(args), directory = "data",
        name = "gmap_tair10chr/", create = FALSE)
    p <- GsnapParam(genome = gmapGenome, unique_only = TRUE,
        molecule = "DNA", max_mismatches = 3)
    o <- gsnap(input_a = infile1(args)[x], input_b = infile2(args)[x],
        params = p, output = outfile1(args)[x])
}
resources <- list(walltime = 120, ntasks = 1, ncpus = cores(args),
    memory = 1024)
param <- BatchtoolsParam(workers = 4, cluster = "slurm", template = "batchtools.slurm.tmpl",
    resources = resources)
d <- bplapply(seq(along = args), f, BPPARAM = param)
```

3.8 Read counting for mRNA profiling experiments

Create `txdb` (needs to be done only once).

```
library(GenomicFeatures)
txdb <- makeTxDbFromGFF(file = "data/tair10.gff", format = "gff",
    dataSource = "TAIR", organism = "Arabidopsis thaliana")
saveDb(txdb, file = "./data/tair10.sqlite")
```

The following performs read counting with `summarizeOverlaps` in parallel mode with multiple cores.

```
library(BiocParallel)
txdb <- loadDb("./data/tair10.sqlite")
eByg <- exonsBy(txdb, by = "gene")
bfl <- BamFileList(outpaths(args), yieldSize = 50000, index = character())
multicoreParam <- MulticoreParam(workers = 4)
```

```r
register(multicoreParam)
registered()
counteByg <- bplapply(bfl, function(x) summarizeOverlaps(eByg,
    x, mode = "Union", ignore.strand = TRUE, inter.feature = TRUE,
    singleEnd = TRUE))

# Note: for strand-specific RNA-Seq set 'ignore.strand=FALSE'
# and for PE data set 'singleEnd=FALSE'
countDFeByg <- sapply(seq(along = counteByg), function(x) assays(counteByg[[x]])$counts)
rownames(countDFeByg) <- names(rowRanges(counteByg[[1]]))
colnames(countDFeByg) <- names(bfl)
rpkmDFeByg <- apply(countDFeByg, 2, function(x) returnRPKM(counts = x,
    ranges = eByg))
write.table(countDFeByg, "results/countDFeByg.xls", col.names = NA,
    quote = FALSE, sep = "\t")
write.table(rpkmDFeByg, "results/rpkmDFeByg.xls", col.names = NA,
    quote = FALSE, sep = "\t")
```

Please note, in addition to read counts this step generates RPKM normalized expression values. For most statistical differential expression or abundance analysis methods, such as `edgeR` or `DESeq2`, the raw count values should be used as input. The usage of RPKM values should be restricted to specialty applications required by some users, *e.g.* manually comparing the expression levels of different genes or features.

Read counting with `summarizeOverlaps` using multiple nodes of a cluster.

```r
library(BiocParallel)
f <- function(x) {
    library(systemPipeR)
    library(BiocParallel)
    library(GenomicFeatures)
    txdb <- loadDb("./data/tair10.sqlite")
    eByg <- exonsBy(txdb, by = "gene")
    args <- systemArgs(sysma = "param/tophat.param", mytargets = "targets.txt")
    bfl <- BamFileList(outpaths(args), yieldSize = 50000, index = character())
    summarizeOverlaps(eByg, bfl[x], mode = "Union", ignore.strand = TRUE,
        inter.feature = TRUE, singleEnd = TRUE)
}
resources <- list(walltime = 120, ntasks = 1, ncpus = cores(args),
    memory = 1024)
param <- BatchtoolsParam(workers = 4, cluster = "slurm", template = "batchtools.slurm.tmpl",
    resources = resources)
counteByg <- bplapply(seq(along = args), f, BPPARAM = param)
countDFeByg <- sapply(seq(along = counteByg), function(x) assays(counteByg[[x]])$counts)
rownames(countDFeByg) <- names(rowRanges(counteByg[[1]]))
colnames(countDFeByg) <- names(outpaths(args))
```

3.9 Read counting for miRNA profiling experiments

Download miRNA genes from miRBase.

```
system("wget ftp://mirbase.org/pub/mirbase/19/genomes/My_species.gff3 -P ./data/")
gff <- import.gff("./data/My_species.gff3")
gff <- split(gff, elementMetadata(gff)$ID)
bams <- names(bampaths)
names(bams) <- targets$SampleName
bfl <- BamFileList(bams, yieldSize = 50000, index = character())
countDFmiR <- summarizeOverlaps(gff, bfl, mode = "Union", ignore.strand = FALSE,
    inter.feature = FALSE)  # Note: inter.feature=FALSE important since pre and mature miRNA ranges overlap
rpkmDFmiR <- apply(countDFmiR, 2, function(x) returnRPKM(counts = x,
    gffsub = gff))
write.table(assays(countDFmiR)$counts, "results/countDFmiR.xls",
    col.names = NA, quote = FALSE, sep = "\t")
write.table(rpkmDFmiR, "results/rpkmDFmiR.xls", col.names = NA,
    quote = FALSE, sep = "\t")
```

3.10 Correlation analysis of samples

The following computes the sample-wise Spearman correlation coefficients from the *rlog* (regularized-logarithm) transformed expression values generated with the *DESeq2* package. After transformation to a distance matrix, hierarchical clustering is performed with the *hclust* function and the result is plotted as a dendrogram (sample_tree.pdf).

```
library(DESeq2, warn.conflicts = FALSE, quietly = TRUE)
library(ape, warn.conflicts = FALSE)
countDFpath <- system.file("extdata", "countDFeByg.xls", package = "systemPipeR")
countDF <- as.matrix(read.table(countDFpath))
colData <- data.frame(row.names = targetsin(args)$SampleName,
    condition = targetsin(args)$Factor)
dds <- DESeqDataSetFromMatrix(countData = countDF, colData = colData,
    design = ~condition)
d <- cor(assay(rlog(dds)), method = "spearman")
hc <- hclust(dist(1 - d))
plot.phylo(as.phylo(hc), type = "p", edge.col = 4, edge.width = 3,
    show.node.label = TRUE, no.margin = TRUE)
```

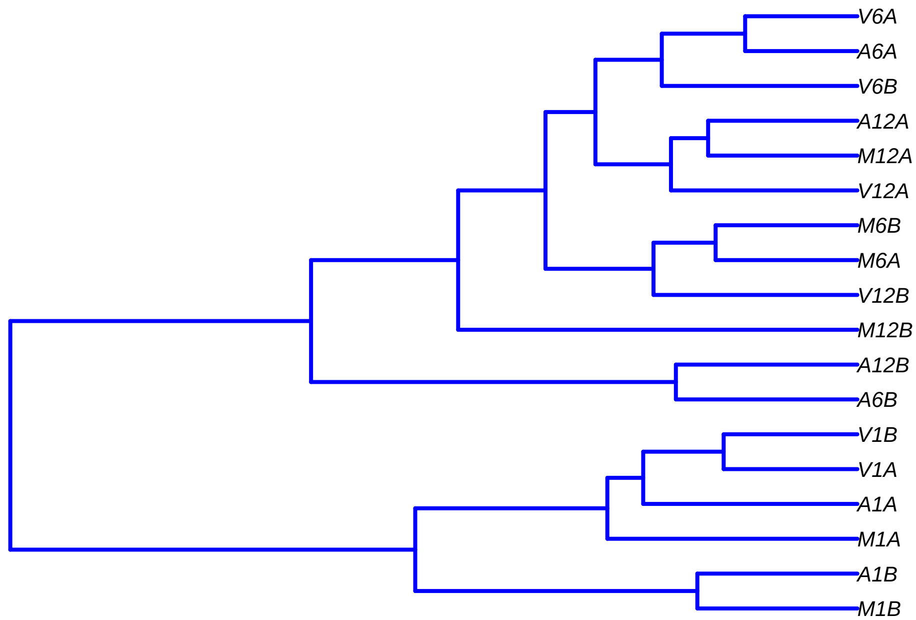

Figure 3: Correlation dendrogram of samples for `rlog` values.

Alternatively, the clustering can be performed with `RPKM` normalized expression values. In combination with Spearman correlation the results of the two clustering methods are often relatively similar.

```
rpkmDFeBygpath <- system.file("extdata", "rpkmDFeByg.xls", package = "systemPipeR")
rpkmDFeByg <- read.table(rpkmDFeBygpath, check.names = FALSE)
rpkmDFeByg <- rpkmDFeByg[rowMeans(rpkmDFeByg) > 50, ]
d <- cor(rpkmDFeByg, method = "spearman")
hc <- hclust(as.dist(1 - d))
plot.phylo(as.phylo(hc), type = "p", edge.col = "blue", edge.width = 2,
    show.node.label = TRUE, no.margin = TRUE)
```

3.11 DEG analysis with `edgeR`

The following `run_edgeR` function is a convenience wrapper for identifying differentially expressed genes (DEGs) in batch mode with `edgeR`'s GML method (Robinson, McCarthy, and Smyth 2010) for any number of pairwise sample comparisons specified under the `cmp` argument. Users are strongly encouraged to consult the `edgeR` vignette for more detailed information on this topic and how to properly run `edgeR` on data sets with more complex experimental designs.

```
targets <- read.delim(targetspath, comment = "#")
cmp <- readComp(file = targetspath, format = "matrix", delim = "-")
cmp[[1]]
##      [,1] [,2]
## [1,] "M1" "A1"
## [2,] "M1" "V1"
## [3,] "A1" "V1"
## [4,] "M6" "A6"
```

```
## [5,] "M6"  "V6"
## [6,] "A6"  "V6"
## [7,] "M12" "A12"
## [8,] "M12" "V12"
## [9,] "A12" "V12"
countDFeBygpath <- system.file("extdata", "countDFeByg.xls",
    package = "systemPipeR")
countDFeByg <- read.delim(countDFeBygpath, row.names = 1)
edgeDF <- run_edgeR(countDF = countDFeByg, targets = targets,
    cmp = cmp[[1]], independent = FALSE, mdsplot = "")
## Disp = 0.21829 , BCV = 0.4672
```

Filter and plot DEG results for up and down regulated genes. Because of the small size of the toy data set used by this vignette, the *FDR* value has been set to a relatively high threshold (here 10%). More commonly used *FDR* cutoffs are 1% or 5%. The definition of '*up*' and '*down*' is given in the corresponding help file. To open it, type `?filterDEGs` in the R console.

```
DEG_list <- filterDEGs(degDF = edgeDF, filter = c(Fold = 2, FDR = 10))
```

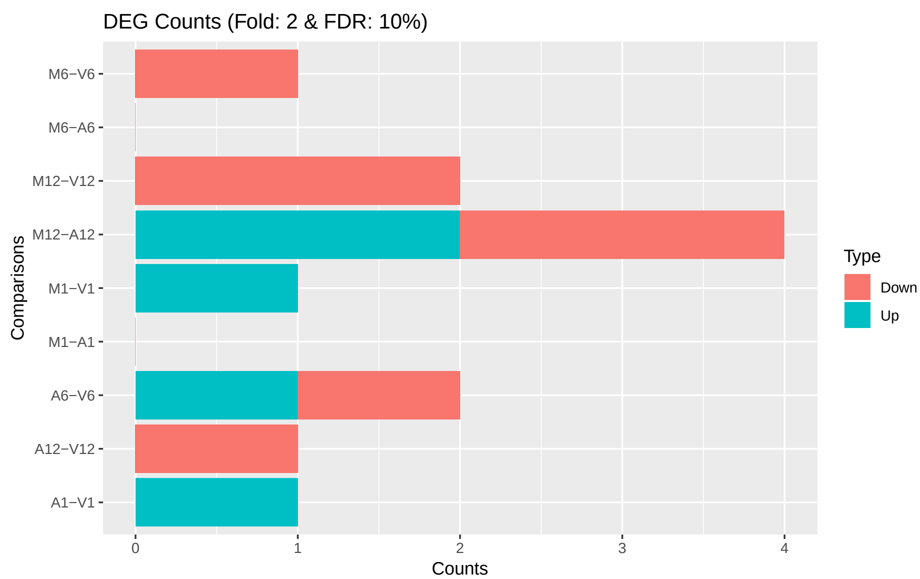

Figure 4: Up and down regulated DEGs identified by `edgeR`.

```
names(DEG_list)
## [1] "UporDown" "Up"       "Down"     "Summary"
DEG_list$Summary[1:4, ]
##        Comparisons Counts_Up_or_Down Counts_Up Counts_Down
## M1-A1        M1-A1                 0         0           0
## M1-V1        M1-V1                 1         1           0
## A1-V1        A1-V1                 1         1           0
## M6-A6        M6-A6                 0         0           0
```

3.12 DEG analysis with *DESeq2*

The following *run_DESeq2* function is a convenience wrapper for identifying DEGs in batch mode with *DESeq2* (Love, Huber, and Anders 2014) for any number of pairwise sample comparisons specified under the *cmp* argument. Users are strongly encouraged to consult the *DESeq2* vignette for more detailed information on this topic and how to properly run *DESeq2* on data sets with more complex experimental designs.

```
degseqDF <- run_DESeq2(countDF = countDFeByg, targets = targets,
    cmp = cmp[[1]], independent = FALSE)
```

Filter and plot DEG results for up and down regulated genes.

```
DEG_list2 <- filterDEGs(degDF = degseqDF, filter = c(Fold = 2,
    FDR = 10))
```

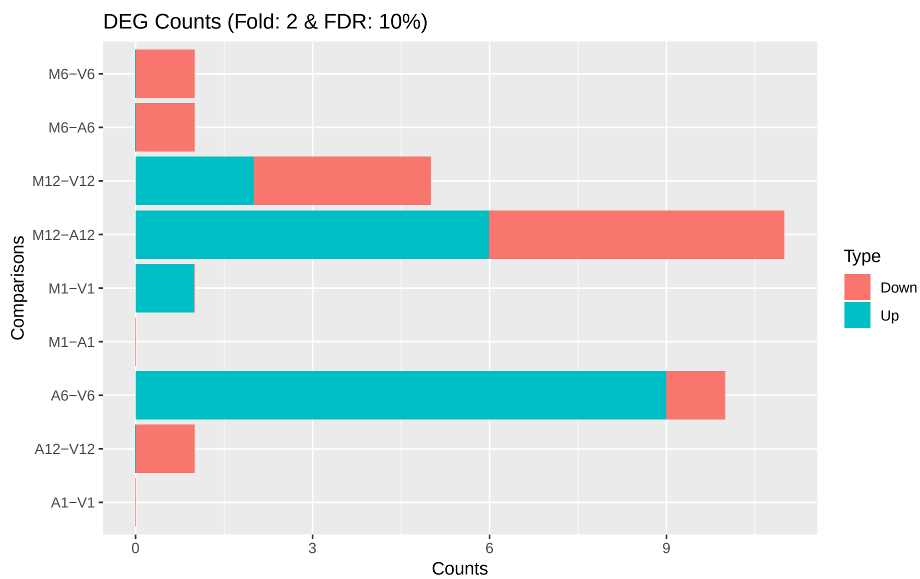

Figure 5: Up and down regulated DEGs identified by *DESeq2*.

3.13 Venn Diagrams

The function *overLapper* can compute Venn intersects for large numbers of sample sets (up to 20 or more) and *vennPlot* can plot 2-5 way Venn diagrams. A useful feature is the possiblity to combine the counts from several Venn comparisons with the same number of sample sets in a single Venn diagram (here for 4 up and down DEG sets).

```
vennsetup <- overLapper(DEG_list$Up[6:9], type = "vennsets")
vennsetdown <- overLapper(DEG_list$Down[6:9], type = "vennsets")
vennPlot(list(vennsetup, vennsetdown), mymain = "", mysub = "",
    colmode = 2, ccol = c("blue", "red"))
```

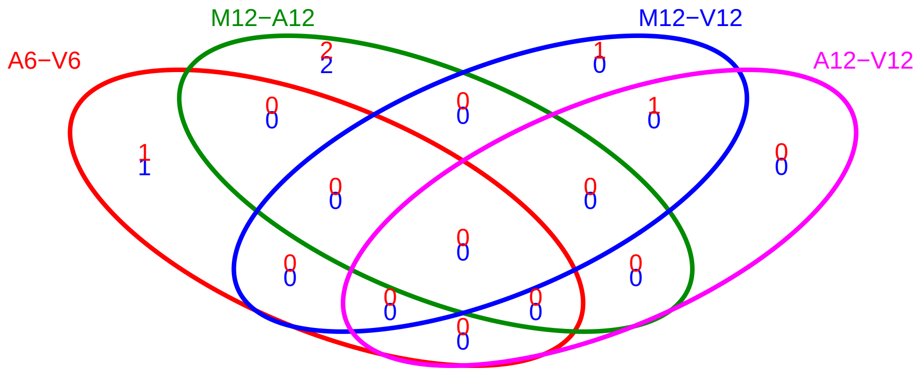

Figure 6: Venn Diagram for 4 Up and Down DEG Sets.

3.14 GO term enrichment analysis of DEGs

3.14.1 Obtain gene-to-GO mappings

The following shows how to obtain gene-to-GO mappings from `biomaRt` (here for *A. thaliana*) and how to organize them for the downstream GO term enrichment analysis. Alternatively, the gene-to-GO mappings can be obtained for many organisms from Bioconductor's `*.db` genome annotation packages or GO annotation files provided by various genome databases. For each annotation this relatively slow preprocessing step needs to be performed only once. Subsequently, the preprocessed data can be loaded with the `load` function as shown in the next subsection.

```
library("biomaRt")
listMarts()  # To choose BioMart database
listMarts(host = "plants.ensembl.org")
m <- useMart("plants_mart", host = "plants.ensembl.org")
listDatasets(m)
m <- useMart("plants_mart", dataset = "athaliana_eg_gene", host = "plants.ensembl.org")
listAttributes(m)  # Choose data types you want to download
go <- getBM(attributes = c("go_id", "tair_locus", "namespace_1003"),
    mart = m)
go <- go[go[, 3] != "", ]
go[, 3] <- as.character(go[, 3])
go[go[, 3] == "molecular_function", 3] <- "F"
go[go[, 3] == "biological_process", 3] <- "P"
go[go[, 3] == "cellular_component", 3] <- "C"
go[1:4, ]
dir.create("./data/GO")
write.table(go, "data/GO/GOannotationsBiomart_mod.txt", quote = FALSE,
    row.names = FALSE, col.names = FALSE, sep = "\t")
catdb <- makeCATdb(myfile = "data/GO/GOannotationsBiomart_mod.txt",
    lib = NULL, org = "", colno = c(1, 2, 3), idconv = NULL)
save(catdb, file = "data/GO/catdb.RData")
```

3.14.2 Batch GO term enrichment analysis

Apply the enrichment analysis to the DEG sets obtained in the above differential expression analysis. Note, in the following example the *FDR* filter is set here to an unreasonably high value, simply because of the small size of the toy data set used in this vignette. Batch enrichment analysis of many gene sets is performed with the `GOCluster_Report` function. When `method="all"`, it returns all GO terms passing the p-value cutoff specified under the `cutoff` arguments. When `method="slim"`, it returns only the GO terms specified under the `myslimv` argument. The given example shows how one can obtain such a GO slim vector from BioMart for a specific organism.

```
load("data/GO/catdb.RData")
DEG_list <- filterDEGs(degDF = edgeDF, filter = c(Fold = 2, FDR = 50),
    plot = FALSE)
up_down <- DEG_list$UporDown
names(up_down) <- paste(names(up_down), "_up_down", sep = "")
up <- DEG_list$Up
names(up) <- paste(names(up), "_up", sep = "")
down <- DEG_list$Down
names(down) <- paste(names(down), "_down", sep = "")
DEGlist <- c(up_down, up, down)
DEGlist <- DEGlist[sapply(DEGlist, length) > 0]
BatchResult <- GOCluster_Report(catdb = catdb, setlist = DEGlist,
    method = "all", id_type = "gene", CLSZ = 2, cutoff = 0.9,
    gocats = c("MF", "BP", "CC"), recordSpecGO = NULL)
library("biomaRt")
m <- useMart("plants_mart", dataset = "athaliana_eg_gene", host = "plants.ensembl.org")
goslimvec <- as.character(getBM(attributes = c("goslim_goa_accession"),
    mart = m)[, 1])
BatchResultslim <- GOCluster_Report(catdb = catdb, setlist = DEGlist,
    method = "slim", id_type = "gene", myslimv = goslimvec, CLSZ = 10,
    cutoff = 0.01, gocats = c("MF", "BP", "CC"), recordSpecGO = NULL)
```

3.14.3 Plot batch GO term results

The `data.frame` generated by `GOCluster_Report` can be plotted with the `goBarplot` function. Because of the variable size of the sample sets, it may not always be desirable to show the results from different DEG sets in the same bar plot. Plotting single sample sets is achieved by subsetting the input data frame as shown in the first line of the following example.

```
gos <- BatchResultslim[grep("M6-V6_up_down", BatchResultslim$CLID),
    ]
gos <- BatchResultslim
pdf("GOslimbarplotMF.pdf", height = 8, width = 10)
goBarplot(gos, gocat = "MF")
dev.off()
goBarplot(gos, gocat = "BP")
goBarplot(gos, gocat = "CC")
```

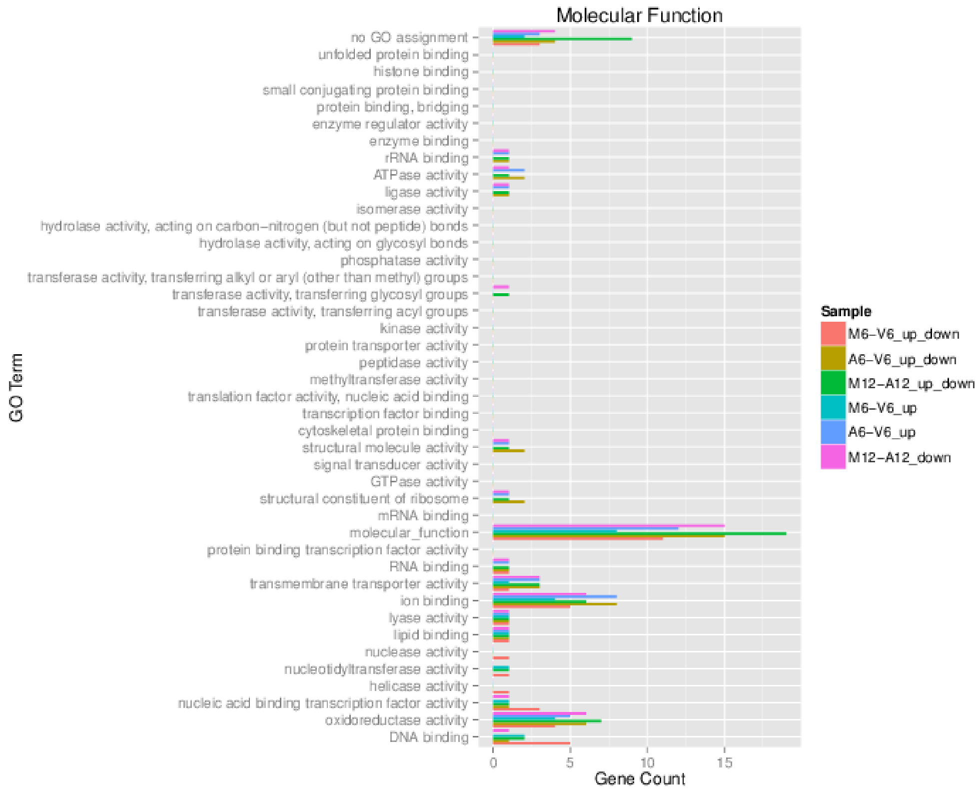

Figure 7: GO Slim Barplot for MF Ontology.

3.15 Clustering and heat maps

The following example performs hierarchical clustering on the `rlog` transformed expression matrix subsetted by the DEGs identified in the above differential expression analysis. It uses a Pearson correlation-based distance measure and complete linkage for cluster joining.

```
library(pheatmap)
geneids <- unique(as.character(unlist(DEG_list[[1]])))
y <- assay(rlog(dds))[geneids, ]
pdf("heatmap1.pdf")
pheatmap(y, scale = "row", clustering_distance_rows = "correlation",
    clustering_distance_cols = "correlation")
dev.off()
```

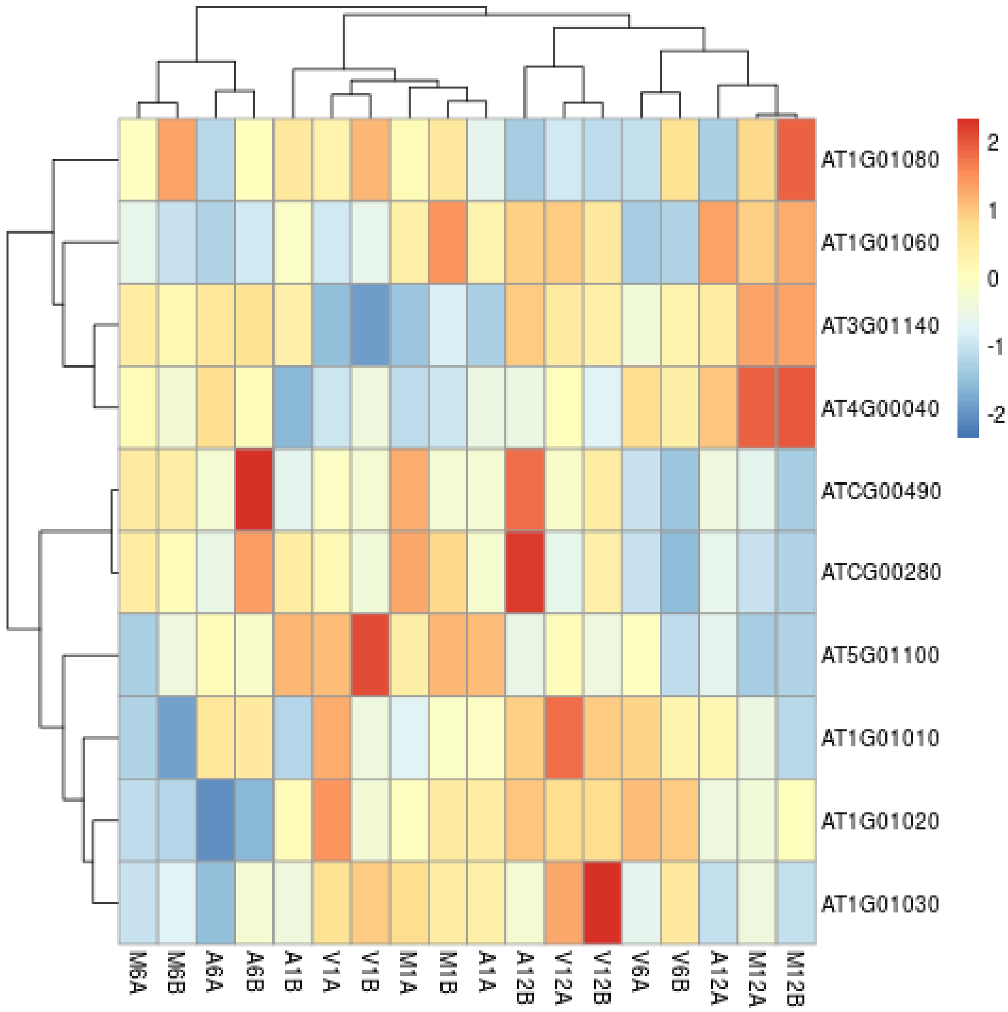

Figure 8: Heat map with hierarchical clustering dendrograms of DEGs.

4 Workflow templates

4.1 RNA-Seq sample

Load the RNA-Seq sample workflow into your current working directory.

```
library(systemPipeRdata)
genWorkenvir(workflow = "rnaseq")
setwd("rnaseq")
```

4.1.1 Run workflow

Next, run the chosen sample workflow `systemPipeRNAseq` (PDF, Rmd) by executing from the command-line `make -B` within the `rnaseq` directory. Alternatively, one can run the code from the provided `*.Rmd` template file from within R interactively.

Workflow includes following steps:

1. Read preprocessing
   - Quality filtering (trimming)
   - FASTQ quality report
2. Alignments: `Tophat2` (or any other RNA-Seq aligner)
3. Alignment stats
4. Read counting
5. Sample-wise correlation analysis
6. Analysis of differentially expressed genes (DEGs)
7. GO term enrichment analysis
8. Gene-wise clustering

4.2 ChIP-Seq sample

Load the ChIP-Seq sample workflow into your current working directory.

```
library(systemPipeRdata)
genWorkenvir(workflow = "chipseq")
setwd("chipseq")
```

4.2.1 Run workflow

Next, run the chosen sample workflow `systemPipeChIPseq_single` (PDF, Rmd) by executing from the command-line `make -B` within the `chipseq` directory. Alternatively, one can run the code from the provided `*.Rmd` template file from within R interactively.

Workflow includes following steps:

1. Read preprocessing
   - Quality filtering (trimming)
   - FASTQ quality report
2. Alignments: `Bowtie2` or `rsubread`
3. Alignment stats
4. Peak calling: `MACS2`, `BayesPeak`
5. Peak annotation with genomic context
6. Differential binding analysis
7. GO term enrichment analysis
8. Motif analysis

4.3 VAR-Seq sample

4.3.1 VAR-Seq workflow for single machine

Load the VAR-Seq sample workflow into your current working directory.

```
library(systemPipeRdata)
genWorkenvir(workflow = "varseq")
setwd("varseq")
```

4.3.2 Run workflow

Next, run the chosen sample workflow `systemPipeVARseq_single` (PDF, Rmd) by executing from the command-line `make -B` within the `varseq` directory. Alternatively, one can run the code from the provided `*.Rmd` template file from within R interactively.

Workflow includes following steps:

1. Read preprocessing
   - Quality filtering (trimming)
   - FASTQ quality report
2. Alignments: `gsnap`, `bwa`
3. Variant calling: `VariantTools`, `GATK`, `BCFtools`
4. Variant filtering: `VariantTools` and `VariantAnnotation`
5. Variant annotation: `VariantAnnotation`
6. Combine results from many samples
7. Summary statistics of samples

4.3.3 VAR-Seq workflow for computer cluster

The workflow template provided for this step is called `systemPipeVARseq.Rmd` (PDF, Rmd). It runs the above VAR-Seq workflow in parallel on multiple computer nodes of an HPC system using Slurm as scheduler.

4.4 Ribo-Seq sample

Load the Ribo-Seq sample workflow into your current working directory.

```
library(systemPipeRdata)
genWorkenvir(workflow = "riboseq")
setwd("riboseq")
```

4.4.1 Run workflow

Next, run the chosen sample workflow `systemPipeRIBOseq` (PDF, Rmd) by executing from the command-line `make -B` within the `ribseq` directory. Alternatively, one can run the code from the provided `*.Rmd` template file from within R interactively.

Workflow includes following steps:

1. Read preprocessing
   - Adaptor trimming and quality filtering
   - FASTQ quality report
2. Alignments: `Tophat2` (or any other RNA-Seq aligner)
3. Alignment stats
4. Compute read distribution across genomic features
5. Adding custom features to workflow (e.g. uORFs)
6. Genomic read coverage along transcripts
7. Read counting
8. Sample-wise correlation analysis
9. Analysis of differentially expressed genes (DEGs)
10. GO term enrichment analysis
11. Gene-wise clustering
12. Differential ribosome binding (translational efficiency)

5 Version information

```
sessionInfo()
## R version 3.5.2 (2018-12-20)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.1 LTS
##
## Matrix products: default
## BLAS: /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.7.1
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.7.1
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats4    parallel  stats     graphics  grDevices
## [6] utils     datasets  methods   base
##
## other attached packages:
##  [1] DESeq2_1.22.1               batchtools_0.9.11
##  [3] data.table_1.12.0           ape_5.2
##  [5] ggplot2_3.1.0               systemPipeR_1.17.5
##  [7] ShortRead_1.40.0            GenomicAlignments_1.18.0
##  [9] SummarizedExperiment_1.12.0 DelayedArray_0.8.0
## [11] matrixStats_0.54.0          Biobase_2.42.0
## [13] BiocParallel_1.16.2         Rsamtools_1.34.0
## [15] Biostrings_2.50.1           XVector_0.22.0
## [17] GenomicRanges_1.34.0        GenomeInfoDb_1.18.1
## [19] IRanges_2.16.0              S4Vectors_0.20.1
```

```
## [21] BiocGenerics_0.28.0         BiocStyle_2.10.0
##
## loaded via a namespace (and not attached):
##   [1] colorspace_1.4-0       rjson_0.2.20
##   [3] hwriter_1.3.2          rprojroot_1.3-2
##   [5] htmlTable_1.13.1       base64enc_0.1-3
##   [7] rstudioapi_0.9.0       bit64_0.9-7
##   [9] AnnotationDbi_1.44.0   codetools_0.2-16
##  [11] splines_3.5.2          geneplotter_1.60.0
##  [13] knitr_1.21             Formula_1.2-3
##  [15] annotate_1.60.0        cluster_2.0.7-1
##  [17] GO.db_3.7.0            pheatmap_1.0.12
##  [19] graph_1.60.0           BiocManager_1.30.4
##  [21] compiler_3.5.2         httr_1.4.0
##  [23] GOstats_2.48.0         backports_1.1.3
##  [25] assertthat_0.2.0       Matrix_1.2-15
##  [27] lazyeval_0.2.1         limma_3.38.2
##  [29] formatR_1.5            acepack_1.4.1
##  [31] htmltools_0.3.6        prettyunits_1.0.2
##  [33] tools_3.5.2            bindrcpp_0.2.2
##  [35] gtable_0.2.0           glue_1.3.0
##  [37] GenomeInfoDbData_1.2.0 Category_2.48.0
##  [39] dplyr_0.7.8            rappdirs_0.3.1
##  [41] Rcpp_1.0.0             nlme_3.1-137
##  [43] rtracklayer_1.42.1     xfun_0.4
##  [45] stringr_1.3.1          XML_3.98-1.16
##  [47] edgeR_3.24.0           zlibbioc_1.28.0
##  [49] scales_1.0.0           hms_0.4.2
##  [51] RBGL_1.58.1            RColorBrewer_1.1-2
##  [53] yaml_2.2.0             gridExtra_2.3
##  [55] memoise_1.1.0          rpart_4.1-13
##  [57] biomaRt_2.38.0         latticeExtra_0.6-28
##  [59] stringi_1.2.4          RSQLite_2.1.1
##  [61] genefilter_1.64.0      checkmate_1.9.1
##  [63] GenomicFeatures_1.34.1 rlang_0.3.1
##  [65] pkgconfig_2.0.2        bitops_1.0-6
##  [67] evaluate_0.12          lattice_0.20-38
##  [69] purrr_0.2.5            bindr_0.1.1
##  [71] labeling_0.3           htmlwidgets_1.3
##  [73] bit_1.1-14             tidyselect_0.2.5
##  [75] GSEABase_1.44.0        AnnotationForge_1.24.0
##  [77] plyr_1.8.4             magrittr_1.5
##  [79] bookdown_0.7           R6_2.3.0
##  [81] Hmisc_4.2-0            base64url_1.4
##  [83] DBI_1.0.0              pillar_1.3.1
##  [85] foreign_0.8-71         withr_2.1.2
##  [87] survival_2.43-3        RCurl_1.95-4.11
##  [89] nnet_7.3-12            tibble_2.0.1
##  [91] crayon_1.3.4           rmarkdown_1.10
##  [93] progress_1.2.0         locfit_1.5-9.1
##  [95] grid_3.5.2             blob_1.1.1
```

```
## [97] Rgraphviz_2.26.0       digest_0.6.18
## [99] xtable_1.8-3           brew_1.0-6
## [101] munsell_0.5.0
```

References

Girke, Thomas. 2014. "systemPipeR: NGS Workflow and Report Generation Environment." UC Riverside. https://github.com/tgirke/systemPipeR.

Howard, Brian E, Qiwen Hu, Ahmet Can Babaoglu, Manan Chandra, Monica Borghi, Xiaoping Tan, Luyan He, et al. 2013. "High-Throughput RNA Sequencing of Pseudomonas-Infected Arabidopsis Reveals Hidden Transcriptome Complexity and Novel Splice Variants." *PLoS One* 8 (10): e74183. doi:10.1371/journal.pone.0074183.

Kim, Daehwan, Ben Langmead, and Steven L Salzberg. 2015. "HISAT: A Fast Spliced Aligner with Low Memory Requirements." *Nat. Methods* 12 (4): 357–60.

Kim, Daehwan, Geo Pertea, Cole Trapnell, Harold Pimentel, Ryan Kelley, and Steven L Salzberg. 2013. "TopHat2: Accurate Alignment of Transcriptomes in the Presence of Insertions, Deletions and Gene Fusions." *Genome Biol.* 14 (4): R36. doi:10.1186/gb-2013-14-4-r36.

Langmead, Ben, and Steven L Salzberg. 2012. "Fast Gapped-Read Alignment with Bowtie 2." *Nat. Methods* 9 (4). Nature Publishing Group: 357–59. doi:10.1038/nmeth.1923.

Lawrence, Michael, Wolfgang Huber, Hervé Pagès, Patrick Aboyoun, Marc Carlson, Robert Gentleman, Martin T Morgan, and Vincent J Carey. 2013. "Software for Computing and Annotating Genomic Ranges." *PLoS Comput. Biol.* 9 (8): e1003118. doi:10.1371/journal.pcbi.1003118.

Li, H, and R Durbin. 2009. "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform." *Bioinformatics* 25 (14): 1754–60. doi:10.1093/bioinformatics/btp324.

Li, Heng. 2013. "Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM." *arXiv [Q-bio.GN]*, March. http://arxiv.org/abs/1303.3997.

Liao, Yang, Gordon K Smyth, and Wei Shi. 2013. "The Subread Aligner: Fast, Accurate and Scalable Read Mapping by Seed-and-Vote." *Nucleic Acids Res.* 41 (10): e108. doi:10.1093/nar/gkt214.

Love, Michael, Wolfgang Huber, and Simon Anders. 2014. "Moderated Estimation of Fold Change and Dispersion for RNA-seq Data with DESeq2." *Genome Biol.* 15 (12): 550. doi:10.1186/s13059-014-0550-8.

Robinson, M D, D J McCarthy, and G K Smyth. 2010. "EdgeR: A Bioconductor Package for Differential Expression Analysis of Digital Gene Expression Data." *Bioinformatics* 26 (1): 139–40. doi:10.1093/bioinformatics/btp616.

Wu, T D, and S Nacu. 2010. "Fast and SNP-tolerant Detection of Complex Variants and Splicing in Short Reads." *Bioinformatics* 26 (7): 873–81. doi:10.1093/bioinformatics/btq057.