flowr

Streamlining Computing Workflows

Sahil Seth

Flowr version 0.9.8.9015 Revised on 2015-11-04

Contents

1 Get started

```
library(flowr)
```

```
setup()
```

This will copy the flowr helper script to `~/bin`. Please make sure that this folder is in your `$PATH` variable. For more details refer to setup's help section.

<!—We have a quite handy command-line-interface for flowr, which exposes all functions of the package to terminal. Such that we dont have to open a interactive R session each time. To make this work, run a setup function which copies the 'flowr' helper script to your `~/bin` directory. —>

Running flowr from the terminal will fetch you the following:

```
Usage: flowr function [arguments]

status          Detailed status of a flow(s).
rerun           rerun a previously failed flow
kill            Kill the flow, upon providing working directory
fetch_pipes     Checking what modules and pipelines are available; flowr fetch_pipes

Please use 'flowr -h function' to obtain further information about the usage of a specific function.
```

1.1 Toy example

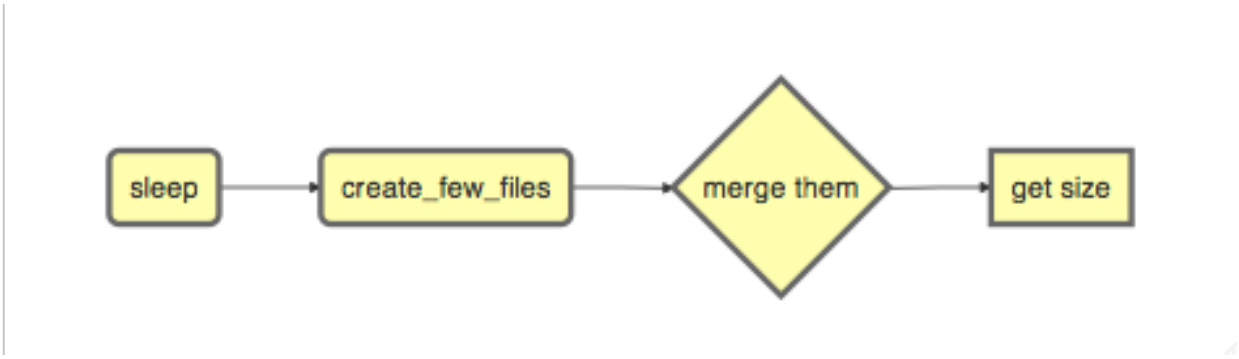

Figure 1:

Consider, a simple example where we have **three** instances of linux's `sleep` command. After its completion **three** tmp files are created with some random data. Then, a merging step follows, combining the tmp files into one big file. Next, we use `du` to calculate the size of the merged file.

NGS context This is quite similar in structure to a typical workflow from where a series of alignment and sorting steps may take place on the raw fastq files. Followed by merging of the resulting bam files into one large file per-sample and further downstream processing.

To create this flow in flowr, we need the actual commands to run; and a set of instructions regarding how to stich the individual steps into a coherent pipeline.

Here is a table with the commands we would like to run (or `flow mat`).

samplename	jobname	cmd
sample1	sleep	sleep 10 && sleep 2;echo hello
sample1	sleep	sleep 11 && sleep 8;echo hello
sample1	sleep	sleep 11 && sleep 17;echo hello
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_1
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_2
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_3
sample1	merge	cat sample1_tmp_1 sample1_tmp_2 sample1_tmp_3 > sample1_merged
sample1	size	du -sh sample1_merged; echo MY shell: $SHELL

Further, we use an additional file specifying the relationship between the steps, and also other resource requirements: flow_def.

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	job
sleep	scatter	none	none	short	2000	1:00	1	local	
create_tmp	scatter	sleep	serial	short	2000	1:00	1	local	
merge	serial	create_tmp	gather	short	2000	1:00	1	local	
size	serial	merge	serial	short	2000	1:00	1	local	

Note: Each row in a flow mat relates to one job. Jobname column is used to link flow definition with flow mat. Also, values in previous jobs (prev_jobs) are derived from jobnames.

1.2 Stitch it

We use the two files described above and stich them to create a `flow object` (which contains all the information we need for cluster submission).

```
ex = file.path(system.file(package = "flowr"), "pipelines")
flowmat = as.flowmat(file.path(ex, "sleep_pipe.tsv"))
flowdef = as.flowdef(file.path(ex, "sleep_pipe.def"))

fobj <- to_flow(x = flowmat,
                def = flowdef,
                flowname = "example1", ## give it a name
                platform = "lsf")      ## override platform mentioned in flow def
```

Refer to to_flow's help section for more details.

1.3 Plot it

We can use `plot_flow` to quickly visualize the flow; this really helps when developing complex workflows.

```
plot_flow(fobj)     # ?plot_flow for more information
plot_flow(flowdef) # plot_flow works on flow definition as well
```

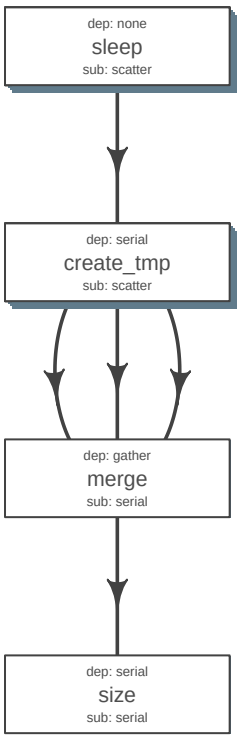

Figure 2: Flow chart describing process for example 1

Refer to plot_flow's help section for more details.

1.4 Dry Run

Dry run: Quickly perform a dry run, of the submission step. This creates all the folder and files, and skips submission to the cluster. This helps in debugging etc.

```
submit_flow(fobj)
```

```
Test Successful!
You may check this folder for consistency. Also you may re-run submit with execute=TRUE
 ~/flowr/sleep_pipe-20150520-15-18-27-5mSd32G0
```

1.5 Submit it

Submit to the cluster !

```
submit_flow(fobj, execute = TRUE)
```

```
Flow has been submitted. Track it from terminal using:
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE
```

Refer to submit_flow's help section for more details.

1.6 Check its status

One may periodically run `status` to monitor the status of a flow.

```
flowr status x=~/flowr/runs/sleep_pipe-20150520*
```

```
|          | total| started| completed| exit_status|    status|
|:---------|-----:|-------:|---------:|-----------:|---------:|
|001.sleep |    10|      10|        10|           0| completed|
|002.tmp   |    10|      10|        10|           0| completed|
|003.merge |     1|       1|         1|           0| completed|
|004.size  |     1|       1|         1|           0| completed|
```

Alternatively, to check a summarized status of several flows, use the parent folder, for example:

```
flowr status x=~/flowr/runs
```

```
Showing status of: ~/flowr/runs
|          | total| started| completed| exit_status|    status|
|:---------|-----:|-------:|---------:|-----------:|---------:|
|001.sleep |    30|      30|        10|           0|processing|
|002.tmp   |    30|      30|        10|           0|processing|
|003.merge |     3|       3|         1|           0|   pending|
|004.size  |     3|       3|         1|           0|   pending|
```

Scalability: Quickly submit, and check a summarized OR detailed status on ten or hundreds of flows.

Refer to status's help section for more details.

1.7 Kill it

Incase something goes wrong, one may use to kill command to terminate all the relating jobs.

kill one flow:

```
flowr kill_flow x=flow_wd
```

One may instruct flowr to kill multiple flows, but flowr would confirm before killing.

```
flowr kill x='~/flowr/runs/sleep_pipe'
found multiple wds:
  ~/flowr/runs/sleep_pipe-20150825-16-24-04-0Lv1PbpI
  ~/flowr/runs/sleep_pipe-20150825-17-47-52-5vFIkrMD
Really kill all of them ? kill again with force=TRUE
```

To kill multiple flow, set force=TRUE:

```
kill(x='~/flowr/runs/sleep_pipe*', force = TRUE)
```

Refer to kill's help section for more details.

1.8 Re-run a flow

flowr also enables you to re-run a pipeline in case of hardware or software failures.

- **hardware failure**: no change to the pipeline is required, simply rerun it: `rerun(x=flow_wd, start_from=<intermediate step>)`
- **software failure**: either a change to flowmat or flowdef has been made: `rerun(x=flow_wd, mat = new_flowmat, def = new_flowdef, start_from=<intermediate step>)`

Refer to rerun's help section for more details.

2 Ingredients for building a pipeline

An easy and quick way to build a workflow is to create a set of two tab delimited files. First is a table with commands to run (for each step of the pipeline), while second has details regarding how the modules are stitched together. In the rest of this document we would refer to them as `flow_mat` and `flow_def` respectively (as introduced in the previous sections).

We could read in, examples of both these files to understand their structure.

```
ex = file.path(system.file(package = "flowr"), "pipelines")
flow_mat = as.flowmat(file.path(ex, "sleep_pipe.tsv"))
flow_def = as.flowdef(file.path(ex, "sleep_pipe.def"))
```

2.1 1. Flow matrix

describes commands to run:

Each row in flow mat describes one shell command, with additional information regarding the name of the step etc.

Essentially, this is a tab delimited file with three columns:

- `samplename`: A grouping column. The table is split using this column and each subset is treated as an individual flow. Thus we may have one flowmat for a series of samples, and the whole set would be submitted as a batch.
  - If all the commands are for a single sample, one can just repeat a dummy name like sample1 all throughout.
- `jobname`: This corresponds to the name of the step. This should match exactly with the jobname column in flow_def table described below.
- `cmd`: A shell command to run. One can get quite creative here. These could be multiple shell commands separated by a `;` or `&&`, more on this here. Though to keep this clean you may just wrap a multi-line command into a script and just source the bash script from here.

Here is an example flow_mat for the flowr described above.

samplename	jobname	cmd
sample1	sleep	sleep 10 && sleep 2;echo hello
sample1	sleep	sleep 11 && sleep 8;echo hello
sample1	sleep	sleep 11 && sleep 17;echo hello
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_1
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_2
sample1	create_tmp	head -c 100000 /dev/urandom > sample1_tmp_3
sample1	merge	cat sample1_tmp_1 sample1_tmp_2 sample1_tmp_3 > sample1_merged
sample1	size	du -sh sample1_merged; echo MY shell: $SHELL

2.2 2. Flow definition

defines how to stich pieces of the (work)flow:

Each row in this table refers to one step of the pipeline. It describes the resources used by the step and also its relationship with other steps, especially, the step immediately prior to it.

It is a tab separated file, with a minimum of 4 columns:

- `jobname`: Name of the step
- `sub_type`: Short for submission type, refers to, how should multiple commands of this step be submitted. Possible values are `serial` or `scatter`.
- `prev_job`: Short for previous job, this would be jobname of the previous job. This can be NA/./none if this is a independent/initial step, and no previous step is required for this to start.
- `dep_type`: Short for dependency type, refers to the relationship of this job with the one defined in `prev_job`. This can take values `none`, `gather`, `serial` or `burst`.

These would be explained in detail, below.

Apart from the above described variables, several others defining the resource requirements of each step are also available. These give great amount of flexibility to the user in choosing CPU, wall time, memory and queue for each step (and are passed along to the HPCC platform).

- `cpu_reserved`
- `memory_reserved`
- `nodes`
- `walltime`
- `queue`

This is especially useful for genomics pipelines, since each step may use different amount of resources. For example, in other frameworks, if one step uses 16 cores these would be blocked and not used during processing of several other steps. Thus resulting in blockage of those cores. Flowr prevents this, by being able to tune resources granurly. Example, one may submit few short steps in `short` queue, and longer steps of the same pipeline in say `long` queue.

Most cluster platforms accept these resource arguments. Essentially a file like this is used as a template, and variables defined in curly braces (ex. `{{{CPU}}}`) are filled up using the flow definition file.

If these (resource requirements) columns are not included in the flow definition, their values should be explicitly defined in the submission template. One may customize the templates as described in the cluster support section.

Here is an example of a typical flow_def file.

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	job
sleep	scatter	none	none	short	2000	1:00	1	local	
create_tmp	scatter	sleep	serial	short	2000	1:00	1	local	
merge	serial	create_tmp	gather	short	2000	1:00	1	local	
size	serial	merge	serial	short	2000	1:00	1	local	

<!— ### Style 2

This style may be more suited for people who like to explore more advanced usage and like to code in R. Also this one find this much faster if jobs and their relationships changes a lot.

Here instead of seperating cmds and definitions one defines them step by step incrementally.

- Use: queue(), to define the computing cluster being used
- Use: multiple calls job()
- Use: flow() to stich the jobs into a flow.

Currently we support LSF, Torque and SGE. Let us use LSF for this example.

```
qobj <- queue(platform = "lsf", queue = "normal", verbose = FALSE)
```

Let us stitch a simple flow with three jobs, which are submitted one after the other.

```
job1 <- job(name = "myjob1", cmds = "sleep1", q_obj = qobj)
job2 <- job(name = "myjob2", cmds = "sleep2", q_obj = qobj, previous_job = "myjob1", dependency_type =
job3 <- job(name = "myjob3", cmds = "sleep3", q_obj = qobj, previous_job = "myjob1", dependency_type =
fobj <- flow(name = "myflow", jobs = list(job1, job2, job3), desc="description")
plot_flow(fobj)
```

The above translates to a flow definition which looks like this:

```
dat <- flowr:::create_jobs_mat(fobj)
knitr:::kable(dat)
```

—>

2.2.1 Example:

Let us use an example flow, to understand submission and dependency types.

Consider three steps A, B and C, where A has 10 commands from A1 to A10, similarly B has 10 commands B1 through B10 and C has a single command, C1. Consider another step D (with D1-D3), which comes after C.

```
step:       A   ----> B  -----> C -----> D
# of cmds  10         10          1         3
```

3 Submission types

This refers to the sub_type column in flow definition.

- `scatter`: submit all commands as parallel, independent jobs.
  - *Submit A1 through A10 as independent jobs*
- `serial`: run these commands sequentially one after the other.
  - *Wrap A1 through A10, into a single job.*

4 Dependency types

This refers to the dep_type column in flow definition.

- `none`: independent job.
  - *Initial step A has no dependency*
- `serial`: *one to one* relationship with previous job.
  - *B1 can start as soon as A1 completes.*
- `gather`: *many to one*, wait for **all** commands in previous job to finish then start the current step.
  - *All jobs of B (1-10), need to complete before C1 is started*
- `burst`: *one to many* wait for the previous step which has one job and start processing all cmds in the current step.
  - *D1 to D3 are started as soon as C1 finishes.*

5 Relationships

Using the above submission and dependency types one can create several types of relationships between former and later jobs. Here are a few examples of relationships one may typically use.

5.1 One to One (serial)

```
               A1 --------> B1
               A2 --------> B1
               .. --------> ..
              A10 --------> B10
dependency submission  dependency submission
   none     scatter      serial     scatter
               relationship
                ONE-to-ONE
```

Relationship between steps A and B is best defined as `serial`. Step A (A1 through A10) is submitted as scatter. Further, i^th jobs of B depends on i^th jobs of A. i.e. B1 requires A1 to complete; B2 requires A2 and so on. Also, we note that defining dependency as serial, makes sure that B does not wait for all elements of A to complete.

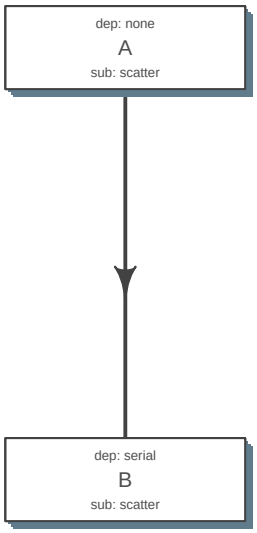

Figure 3:

5.2 Many to One (gather)

```
               B1 ----\
               B2 -----\
               ..       -----> C1
               B9 ------/
               B10-----/
dependency submission  dependency submission
   serial     scatter    gather     serial
               relationship
                MANY-to-ONE
```

Since C is a single command which requires all steps of B to complete, intuitively it needs to `gather` pieces of data generated by B. In this case `dep_type` would be `gather` and `sub_type` type would be `serial` since it is a single command.

<!— - makes sense when previous job had many commands running in parallel and current job would wait for all - so previous job submission: `scatter`, and current job's dependency type `gather`

—>

5.3 One to Many (Burst)

```
                    /-----> D1
                C1 --------> D2
                    \-----> D3
 dependency submission  dependency submission
    gather   serial       burst     scatter
                relationship
                 ONE-to-MANY
```

Further, D is a set of three commands (D1-D3), which need to wait for a single process (C1) to complete. They would be submitted as `scatter` after waiting on C in a `burst` type dependency.

<!— - makes sense when previous job had one command current job would split and submit several jobs in parallel - so previous job submission_type: `serial`, and current job's dependency type `burst`, with a submission type: `scatter`

—>

In essence, an example flow_def would look like as follows (with additional resource requirements not shown for brevity):

```
ex2def = as.flowdef(file.path(ex, "abcd.def"))
ex2mat = as.flowmat(file.path(ex, "abcd.tsv"))
kable(ex2def[, 1:4])
```

jobname	sub_type	prev_jobs	dep_type
A	scatter	none	none
B	scatter	A	serial
C	serial	B	gather
D	scatter	C	burst

```
plot_flow(ex2def)
```

There is a darker more prominent shadow to indicate scatter steps.

6 Cluster Support

As of now we have tested this on the following clusters:

Platform	command	status	queue.type
LSF 7	bsub	Beta	lsf
LSF 9.1	bsub	Stable	lsf
Torque	qsub	Stable	torque
Moab	msub	Stable	moab
SGE	qsub	Beta	sge
SLURM	sbatch	alpha	slurm

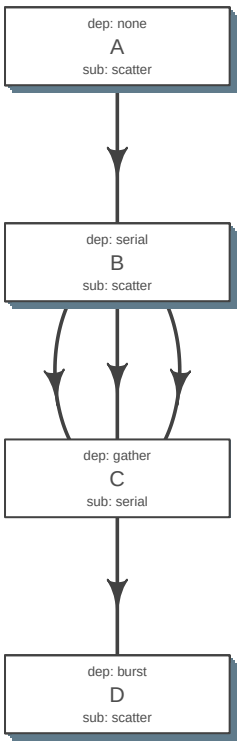

Figure 4:

For more details, refer to the configuration section

7 Installation

Requirements:

- R version > 3.1, preferred 3.2

```
## for a latest official version (from CRAN)
install.packages("flowr", repos = CRAN="http://cran.rstudio.com")

## Latest stable release from DRAT (updated every other week); CRAN for dependencies
install.packages("flowr", repos = c(CRAN="http://cran.rstudio.com", DRAT="http://sahilseth.github.io/dra

## OR cutting edge devel version
devtools::install_github("sahilseth/flowr", ref = "devel")
```

After installation run `setup()`, this will copy the flowr's helper script to `~/bin`. Please make sure that this folder is in your `$PATH` variable.

```
library(flowr)
setup()
```

Running `flowr` from the terminal should now show the following:

```
Usage: flowr function [arguments]

status          Detailed status of a flow(s).
rerun           rerun a previously failed flow
kill            Kill the flow, upon providing working directory
fetch_pipes     Checking what modules and pipelines are available; flowr fetch_pipes

Please use 'flowr -h function' to obtain further information about the usage of a specific function.
```

If you interested, visit funr's github page for more details

From this step on, one has the option of typing commands in a R console OR a bash shell (command line). For brevity we will show examples using the shell.

8 Test

Test a small pipeline on the cluster

This will run a three step pipeline, testing several different relationships between jobs. Initially, we can test this locally, and later on a specific HPCC platform.

```
## This may take about a minute or so.
flowr run x=sleep_pipe platform=local execute=TRUE
## corresponding R command:
run(pipe='sleep_pipe', platform='local', execute=TRUE)
```

If this completes successfully, we can try this on a computing cluster; where this would submit a few interconnected jobs.

Several platforms are supported out of the box (torque, moab, sge, slurm and lsf), you may use the platform variable to switch between platforms.

```
flowr run pipe=sleep_pipe platform=lsf execute=TRUE
## other options for platform: torque, moab, sge, slurm, lsf
## this shows the folder being used as a working directory for this flow.
```

Once the submission is complete, we can test the status using `status()` by supplying it the full path as recovered from the previous step.

```
flowr status x=~/flowr/runs/sleep_pipe-samp1-20150923-10-37-17-4WBiLgCm

## we expect to see a table like this when is completes successfully:

|               | total| started| completed| exit_status|status    |
|:--------------|-----:|-------:|---------:|-----------:|:---------|
|001.sleep      |     3|       3|         3|           0|completed |
|002.create_tmp |     3|       3|         3|           0|completed |
|003.merge      |     1|       1|         1|           0|completed |
|004.size       |     1|       1|         1|           0|completed |

## Also we expect a few files to be created:
ls ~/flowr/runs/sleep_pipe-samp1-20150923-10-37-17-4WBiLgCm/tmp
samp1_merged  samp1_tmp_1  samp1_tmp_2  samp1_tmp_3

## If both these checks are fine, we are all set !
```

There are a few places where things may go wrong, you may follow the advanced configuration guide for more details. Feel free to post questions on github issues page.

9 Advanced Configuration

9.1 HPCC Support Overview

Support for several popular cluster platforms is built-in. There is a template, for each platform, which should work out of the box. Further, one may copy and edit them (and save to `~/flowr/conf`) in case some changes are required. Templates from this folder (`~/flowr/conf`), would override defaults.

Here are links to latest templates on github:

- torque
- lsf
- moab
- sge
- slurm, needs testing

Not sure what platform you have?

You may check the version by running ONE of the following commands:

```
msub --version
## Version: **moab** client 8.1.1
man bsub
##Submits a job to **LSF** by running the specified
qsub --help
```

Here are some helpful guides and details on the platforms:

- PBS: wiki
- Torque: wiki
  - MD Anderson
  - University of Houston
- LSF wiki:
  - Harvard Medicla School uses: LSF HPC 7
  - Also used at Broad
- SGE wiki
  - A tutorial for Sun Grid Engine
  - Another from JHSPH
  - Dependecy info here

Comparison_of_cluster_software

10 Troubleshooting & FAQs

10.1 Errors in job submission

Possible issue: Jobs are not getting submitted

1. Check if the right platform was used for submission.
2. Confirm (with your system admin) that you have the privilege to submit jobs.
3. **Use a custom flowdef**: Many institutions have strict specification on the resource reservations. Make sure that the queue, memory, walltime, etc. requiremets are specified properly
4. **Use a custom submission template**: There are several parameters in the submission script used to submit jobs to the cluster. You may customize this template to suit your needs.

3. Use a custom flowdef

We can copy an example flow definition and customize it to suit our needs. This a tab delimited text file, so make sure that the format is correct after you make any changes.

```
cd ~/flowr/pipelines
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/pipelines/sleep_pipe.def
## check the format
flowr as.flowdef x=~/flowr/pipelines/sleep_pipe.def
```

Run the test with a custom flowdef:

```
flowr run x=sleep_pipe execute=TRUE def=~/flowr/pipelines/sleep_pipe.def ## platform=lsf [optional, pic
```

4. Use a custom submission template

If you need to customize the HPCC submission template, copy the file for your platform and make your desired changes. For example the MOAB based cluster in our institution does **not** accept the `queue` argument, so we need to comment it out.

Download the template for a specific HPCC platform into `~/flowr/conf`

```
cd ~/flowr/conf ## flowr automatically picks up a template from this folder.
## for MOAB (msub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/moab.sh
## for Torque (qsub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/torque.sh
## for IBM LSF (bsub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/lsf.sh
## for SGE (qsub)
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/sge.sh
## for SLURM (sbatch) [untested]
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/slurm.sh
```

Make the desired changes using your favourite editor and submit again.

Possible issue: Jobs for subsequent steps are not submitting (though first step works fine).

1. **Confirm jobids are parsing fine**: Flowr parses the computing platform's output and extracts job IDs of submitted jobs.

2. **Check dependency string**:

1. Parsing job ids

Flowr parses job IDs to keep a log of all submitted jobs, and also to pass them along as a dependency to subsequent jobs. This is taken care by the parse_jobids() function. Each job scheduler shows the jobs id, when you submit a job, but it may show it in a slightly different fashion. To accommodate this one can use regular expressions as described in the relevant section of the flowr config.

For example LSF may show a string such as:

```
Job <335508> is submitted to queue <transfer>.
## test if it parses correctly
jobid="Job <335508> is submitted to queue <transfer>."
set_opts(flow_parse_lsf = ".*(\<[0-9]*\>).*")
parse_jobids(jobid, platform="lsf")
[1] "335508"
```

In this case *335508* was the job id and regex worked well !

Once we identify the correct regex for the platform you may update the configuration file with it.

```
cd ~/flowr/conf
wget https://raw.githubusercontent.com/sahilseth/flowr/master/inst/conf/flowr.conf
## flowr automatically reads from this location, if you prefer to put it elsewhere, use
load_opts("flowr.conf") ## visit sahilseth.github.io/params for more details.
```

Update the regex pattern and submit again.

2. Check dependency string

After collecting job ids from previous jobs, flowr renders them as a dependency for subsequent jobs. This is handled by render_dependency.PLATFORM functions.

Confirm that the dependency parameter is specified correctly in the submission scripts:

```
wd=~/flowr/runs/sleep_pipe-samp1-20150923-11-20-39-dfvhp5CK ## path to the most recent submission
cat $wd/002.create_tmp/create_tmp_cmd_1.sh
```

10.1.0.1 Flowr Configuration file

Possible issue: Flowr shows too much OR too little information.

There are several verbose levels available (0, 1, 2, 3, . . .)

One can change the verbose levels in this file (`~/flowr/conf/flowr.conf`) and check verbosity section in the help pages for more details.

10.2 Flowdef resource columns

Possible issue: What all resources are supported in the flow definition?

The resource requirement columns of flow definition are passed along to the final (cluster) submission script. For example values in `cpu_reserved` column would be populated as `{{{CPU}}}` in the submission template.

The following table provides a mapping between the flow definition columns and variables in the submission templates:

flowdef variable	submission template variable
nodes	NODES
cpu_reserved	CPU
memory_reserved	MEMORY
email	EMAIL
walltime	WALLTIME
extra_opts	EXTRA_OPTS
	JOBNAME
	STDOUT
	CWD
	DEPENDENCY
	TRIGGER
	CMD

`* These are generated on the fly` and `** This is gathered from flow mat`

10.3 Adding a new platform

Possible issue: Need to add a new platform

Adding a new platform involves a few steps, briefly we need to consider the following steps where changes would be necessary.

1. **job submission**: One needs to add a new template for the new platform. Several examples are available as described in the previous section.
2. **parsing job ids**: flowr keeps a log of all submitted jobs, and also to pass them along as a dependency to subsequent jobs. This is taken care by the parse_jobids() function. Each job scheduler shows the jobs id, when you submit a job, but each shows it in a slightly different pattern. To accommodate this one can use regular expressions as described in the relevant section of the flowr config.
3. **render dependency**: After collecting job ids from previous jobs, flowr renders them as a dependency for subsequent jobs. This is handled by render_dependency.PLATFORM functions.
4. **recognize new platform**: Flowr needs to be made aware of the new platform, for this we need to add a new class using the platform name. This is essentially a wrapper around the job class

Essentially this requires us to add a new line like: `setClass("torque", contains = "job")`.

5. **killing jobs**: Just like submission flowr needs to know what command to use to kill jobs. This is defined in detect_kill_cmd function.

There are several job scheduling systems available and we try to support the major players. Adding support is quite easy if we have access to them. Your favourite not in the list? re-open this issue, with details on the platform: adding platforms

Possible issue: For other issues upload the error shown in the out files to github issues tracker.

```
## outfiles end with .out, and are placed in a folder like 00X.<jobname>/
## here is one example:
cat $wd/002.create_tmp/create_tmp_cmd_1.out
## final script:
cat $wd/002.create_tmp/create_tmp_cmd_1.sh
```

10.4 Installation Error (DRAT)

```
install.packages("flowr", repos = "http://sahilseth.github.io/drat")

ERROR: dependency ‘whisker’ is not available for package ‘params’
* removing ‘/usr/local/lib/R/site-library/params’
ERROR: dependencies ‘params’, ‘diagram’, ‘whisker’ are not available for package ‘flowr’
* removing ‘/usr/local/lib/R/site-library/flowr’

The downloaded source packages are in
    ‘/tmp/RtmpykBS2r/downloaded_packages’
Warning messages:
1: In install.packages("flowr", repos = "http://sahilseth.github.io/drat") :
  installation of package ‘params’ had non-zero exit status
2: In install.packages("flowr", repos = "http://sahilseth.github.io/drat") :
  installation of package ‘flowr’ had non-zero exit status
```

Issues is that whisker and params are not installed, and are not available in the DRAT repo.

Solution 1:

```
install.packages("whisker")
install.packages("diagram")
install.packages("flowr", repos = "http://sahilseth.github.io/drat")
```

Solution 2:

```
install.packages("flowr", repos = c(CRAN = "http://cran.rstudio.com", DRAT = "http://sahilseth.github.io
```

10.5 Installation Error (Github)

```
devtools:::install_github("sahilseth/flowr")
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
```

Solution:

This is basically a issue with httr (link) Try this:

```
install.packages("RCurl")
devtools:::install_github("sahilseth/flowr")
```

If not then try this: install.packages(“httr”);

```
library(httr);
set_config( config( ssl.verifypeer = 0L ) )
devtools:::install_github("sahilseth/flowr")
```

11 Creating input file(s)

Let us use the same example described in the overview section. We start by getting a set of commands we would like to run.

```
## wait for a few seconds...
sleep 5
sleep 5

## create two small files
cat $RANDOM > tmp1
cat $RANDOM > tmp2

## merge the two files
cat tmp1 tmp2 > tmp

## check the size of the resulting file
du -sh tmp
```

Wrap these commands into R

```
sleep=c('sleep 5', 'sleep 5')

tmp=c('cat $RANDOM > tmp1',
            'cat $RANDOM > tmp2')

merge='cat tmp1 tmp2 > tmp'

size='du -sh tmp'
```

Next, we would create a table using the above commands:

```
## create a table of all commands
library(flowr)
lst = list( sleep=sleep,
           create_tmp=tmp,
           merge=merge,
           size=size)

flowmat = to_flowmat(lst, "samp1")
kable(flowmat)
```

samplename	jobname	cmd
samp1	sleep	sleep 5
samp1	sleep	sleep 5
samp1	create_tmp	cat $RANDOM > tmp1
samp1	create_tmp	cat $RANDOM > tmp2
samp1	merge	cat tmp1 tmp2 > tmp
samp1	size	du -sh tmp

11.1 Creating Flow Definition

We have a few steps in a pipeline; we would use a flow definition to descibe their flow. Flowr enables us to quickly create a skeleton flow definition using a flowmat, which we can then alter to suit our needs. A handy function `to_flowdef`, accepts a `flowmat` and creates a flow definition.

```
## create a skeleton flow definition
def = to_flowdef(flowmat)
suppressMessages(plot_flow(def))
```

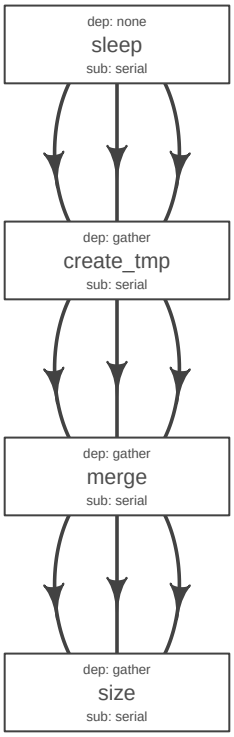

Figure 5:

The default skeleton takes a very conservative approach, creating all submissions as **serial** and all dependencies as **gather**. This ensures robustness, compromising efficiency. So customize this to make it super efficient.

We can make a few changes to make this pipeline a little more efficient. Briefly, we would run a few steps in a **scatter** fashion (in parallel).

A few points to note:

- Intial steps have no dependency, so their **previous_jobs** and **dependency_type** is **none**.
- Steps with multiple commands, which can be run in parallel are submitted as **scatter**.
- Steps with single commands are submitted as **serial**.
- Say two consective steps run on small pieces of data, we have a **serial one to one** relationship. Example, both **sleep** and **create_tmp** are submitted as **scatter** and **create_tmp** has a dependency_type **serial**.
- Finally if a step needs all the small pieces from a previous step, we use a **gather** type dependency.

```
##               sleep     create tmp   merge     size
def$sub_type = c("scatter", "scatter", "serial", "serial")
def$dep_type = c("none", "serial", "gather", "serial")
kable(def)
```

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	job
sleep	scatter	none	none	short	2000	1:00	1	torque	
create_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	
merge	serial	create_tmp	gather	short	2000	1:00	1	torque	
size	serial	merge	serial	short	2000	1:00	1	torque	

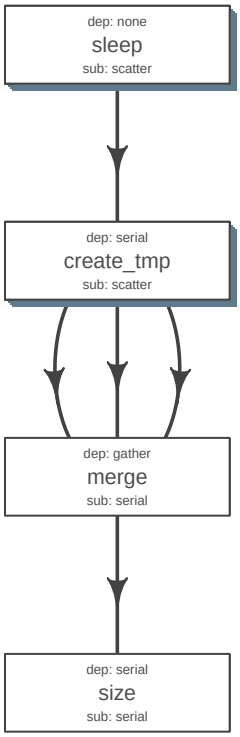

Figure 6:

Tip Alternatively, one may write this to a file (**write_sheet(def, "sleep_pipe.def")**), make changes in a text editor and read it again (**as.flowdef("sleep_pipe.def")**.

11.2 Create flow, submit to cluster

Next, we create a flow object:

```
fobj = to_flow(flowmat, def, flowname = "sleep_pipe")
```

Finally, we can submit this to the cluster:

```
plot_flow(fobj)
submit_flow(fobj) ## dry run
fobj2 = submit_flow(fobj, execute = TRUE) ## submission to LSF cluster

## after submission, we can use the following:
status(fobj2) ## check status
rerun(fobj2)  ## re-run from a intermediate step
kill(fobj2)   ## kill it!
```

11.3 Creating modules

We used a simple example where a single function was creating all the commands. This is easier, but a step (or module) is not re-usable in another pipeline. Thus we may write a module for each step, such that one may mix and match to create their own pipeline.

NOTE: A module, always returns a flowmat. A module may have one or several steps. A module + flowdef, becomes a pipeline.

```
## to follow this tutorial, you may download them:
url=https://raw.githubusercontent.com/sahilseth/flowr/master/inst/pipelines
cd ~/flowr/pipelines
wget $url/sleep_pipe.R ## A R script, with sleep_pipe(), which creates a flowmat
wget $url/sleep_pipe.def ## A tab-delimited flow definition file
wget $url/sleep_pipe.conf ## An *optional* tab-delim conf file, defining default params
```

The `sleep_pipe` calls the three other functions (**modules**); fetches flowmat from each, then rbinds them, creating a larger flowmat. You may refer to the sleep_pipe.R file for the source.

```
#' @param x number of sleep commands
sleep <- function(x, samplename){
    cmd = list(sleep = sprintf("sleep %s && sleep %s;echo 'hello'",
        abs(round(rnorm(x)*10, 0)),
        abs(round(rnorm(x)*10, 0))))
    flowmat = to_flowmat(cmd, samplename)
    return(list(flowmat = flowmat))
}

#' @param x number of tmp commands
create_tmp <- function(x, samplename){
    ## Create 100 temporary files
    tmp = sprintf("%s_tmp_%s", samplename, 1:x)
    cmd = list(create_tmp = sprintf("head -c 100000 /dev/urandom > %s", tmp))
    ## --- convert the list into a data.frame
    flowmat = to_flowmat(cmd, samplename)
    return(list(flowmat = flowmat, outfiles = tmp))
}

#' @param x vector of files to merge
merge_size <- function(x, samplename){
    ## Merge them according to samples, 10 each
    mergedfile = paste0(samplename, "_merged")
    cmd_merge <- sprintf("cat %s > %s",
        paste(x, collapse = " "), ## input files
        mergedfile)
    ## get the size of merged files
    cmd_size = sprintf("du -sh %s; echo 'MY shell:' $SHELL", mergedfile)

    cmd = list(merge = cmd_merge, size = cmd_size)
    ## --- convert the list into a data.frame
    flowmat = to_flowmat(cmd, samplename)
    return(list(flowmat = flowmat, outfiles = mergedfile))
}
```

```
#' @param x number of files to make
sleep_pipe <- function(x = 3, samplename = "samp1"){

    ## call the modules one by one...
    out_sleep = sleep(x, samplename)
    out_create_tmp = create_tmp(x, samplename)
    out_merge_size = merge_size(out_create_tmp$outfiles, samplename)

    ## row bind all the commands
    flowmat = rbind(out_sleep$flowmat,
        out_create_tmp$flowmat,
        out_merge_size$flowmat)

    return(list(flowmat = flowmat, outfiles = out_merge_size$outfiles))
}
```

12 Execute the pipeline

Using run

One may use `run` function to create the flowmat, fetch the flowdef and execute the pipeline in a single step. Here we would focus more on each of these steps in detail.

```
## 1. Single step submission:
fobj = run("sleep_pipe", execute = TRUE);

## 2
## change wd, so that we can source the files downloaded in the previous step
setwd("~/flowr/pipelines")

## 2a. optionally, load default parameters
load_opts("sleep_pipe.conf")

## 2b. get sleep_pipe() function
source("sleep_pipe.R")
## create a flowmat
flowmat = sleep_pipe()

## 2c. read a flow definition.
flowdef = as.flowdef("sleep_pipe.def")

## 2d. create flow and submit to cluster
fobj = to_flow(flowmat, flowdef, execute = TRUE)
```

13 Best practices for writing modules/pipelines

These are some of the practices we follow in-house. We feel using these makes stitching custom pipelines using a set of modules quite easy. Consider this a check-list of a few ideas and a work in progress.

13.1 A note on module functions

```
picard_merge <- function(x,
                         samplename = get_opts("samplename"),
                          mergedbam,
                          java_exe = get_opts("java_exe"),
                          java_mem = get_opts("java_mem"),
                          java_tmp = get_opts("java_tmp"),
                          picard_jar = get_opts("picard_jar")){
    ## Make sure all args have a value (not null)
    ## If a variable was not defined in a conf. file get_opts, will return NULL
    check_args()

  bam_list = paste("INPUT=", x, sep = "", collapse = " ")
  ## create a named list of commands
  cmds = list(merge = sprintf("%s %s -Djava.io.tmpdir=%s -jar %s MergeSamFiles %s OUTPUT=%s ASSUME_SORTE

  ## Create a flowmat
  flowmat = to_flowmat(cmds, samplename)

  ## return a list, flowmat AND outfiles
  return(list(outfiles = mergedbam, flowmat = flowmat))
}
```

1. should accept minimum of **two inputs**,
    - **x** (a input file etc, depends on the module) and
    - samplename (is used to append a column to the flowmat)
2. should always return a list arguments:
    - **flowmat** (required) : contains all the commands to run
    - **outfiles** (recommended): could be used as an input to other tools
3. can define all other default arguments such as paths to tools etc. in a seperate conf (tab-delimited) file.

- Then use `get_opts("param")` to use their value.

```
## Example conf file:
cat my.conf
bwa_exe /apps/bwa/bin/bwa
```

4. should use `check_args()` to make sure none of the default parameters are null.

```
## check_args(), checks ALL the arguments of the function, and throws a error. use ?check_args for more
get_opts("my_new_tool")
```

```
## NULL
```

13.2 Pipeline structure

For example we have a pipeline consisting of alignment using bwa (aln1, aln2, sampe), fix rg tags using picard and merging the files. We would create three files:

```
fastq_bam_bwa.R      ## A R script, with sleep_pipe(), which creates a flowmat
fastq_bam_bwa.conf   ## An *optional* tab-delim conf file, defining default params
fastq_bam_bwa.def    ## A tab-delimited flow definition file
```

Notice how all files have the same basename; this is essential for the **run** function to find all these files.

1. all three files should have the same basename

Reason for using the same basename:

- When we call `run("fastq_bam_bwa", ....)` it tries to look for a .R file inside flowr's package, `~/flowr/pipelines` OR your current wd. If there are multiple matches, later is chosen.
- Then, it finds and load default parameters from `fastq_bam_bwa.conf` (if available).
- Further, it calls the function `fastq_bam_bwa`, then stiches a flow using `fastq_bam_bwa.def` as the flow definition.

2. can have multiple flowdefs like fastq_bam_bwa_lsf.def, fastq_bam_bwa_lsf.def etc, where .def is used by default. But other are available for users to switch platforms quickly.

Feature:

- A user can supply a custom flow definition

```
run('fastq_bam_bwa', def = 'path/myflowdef.def'....)
```

- Starting flowr version *0.9.8.9011*, run also accepts a custom conf file in addition to a flowdef file. Conf contains all the default parameters like absolute paths to tools, paths to genomes, indexes etc.

```
run('fastq_bam_bwa', def = 'path/myflowdef.def', conf='path/myconf.conf',....)
```

This is quite useful for portability, since to use the same pipeline across institution/computing clusters one only needs to change the flow definition and R function remains intact.

Refer to help section on run for more details.

Tip: Its important to note, that in this example we are using R functions, but any other language can be used to create a tab-delimited flowmat file, and submitted using `submit_flow` command.

13.3 Nomeclature for parameters

Here is a good example: https://github.com/sahilseth/flowr/blob/master/inst/pipelines/fastq_bam_bwa.conf

(recommeded for increased compatibility)

1. all binaries end with **__exe**
2. all folders end with **__dir**
3. all jar files end with **__jar**
4. specify cpu's using `<%CPU%>`, this makes this value dynamic and is picked up by the flow definition

<!— output: packagedocs::package_docs packagedocs::package_docs: self_contained: true toc: true —>

14 Example: Fastq to merged BAM

You may visit docs.flowr.space for more details.

If you face any issues, please feel free to raise a issue on github.

14.1 Setup up flowr

Requirements:

- R version > 3.1, preferred 3.2
- install flowr from sahilseth.github.io/drat, provides a more recent version than CRAN.

```
#install.packages("params", repos = "http://cran.rstudio.com")
## for a latest stable version (updated every few days):
install.packages("flowr", repos = "http://sahilseth.github.io/drat")
```

After installation run setup(), this will copy the flowr's helper script to ~/bin. Please make sure that this folder is in your $PATH variable.

```
library(flowr)
setup()
```

```
## Warning in file.copy(confs, flow_conf_path, overwrite = FALSE): problem
## copying /Library/Frameworks/R.framework/Versions/3.2/Resources/library/
## ngsflows/conf/ngsflows.conf to /Users/sahilseth/flowr/conf/ngsflows.conf:
## No such file or directory
```

Then we need to test whether we are able to submit jobs to the cluster properly.

```
## run a test on the local platform
run(x='sleep_pipe', platform='local', execute=FALSE)
## run a test on the HPCC platform (torque, sge, moab, slurm, lsf)
run(x='sleep_pipe', platform='torque', execute=TRUE)
```

NOTE: In case the test is not successful, please follow the advanced configuration page for more details on how to solve the issues.

14.1.1 Fetch and download the pipeline

Next, we will download a pipeline which processes multiple fastq files of a sample into a single aligned and merged BAM file.

```
cd ~/flowr/pipelines
base=https://raw.githubusercontent.com/sahilseth/flowr/devel/inst/pipelines
wget $base/fastq_bam_bwa.R
wget $base/fastq_bam_bwa.conf
wget $base/fastq_bam_bwa.def
```

Reference Genome:

One can download the reference genome including indexes of various alignment tools from Illumina's iGenomes website.

You may skip this step, if you already have the genome fasta and related files.

```
mkdir ~/flowr/genomes; cd ~/flowr/genomes
url=ussd-ftp.illumina.com/Homo_sapiens/NCBI/build37.2/Homo_sapiens_NCBI_build37.2.tar.gz
ftp ftp://igenome:G3nom3s4u@$url
tar -zxvf Homo_sapiens_NCBI_build37.2.tar.gz
```

A typical NGS pipeline consists of many steps, each with several parameters. You can modify `fastq_bam_bwa.conf`, specifying paths to various tools and their default options (samtools, bwa, picard and reference genome indexes).

Note: All parameters of this pipeline are conveniently specified in a tab-delimited format in the **fastq_bam_bwa.conf** file.

```
## customize parameters, including paths to samtools, bwa, reference genomes etc.
vi fastq_bam_bwa.conf
```

Example data:

You may skip this step if you already have raw reads for a sample, in fastq format.

```
mkdir ~/flowr/genomes; cd ~/flowr/genomes
## for testing puposes one may download example fastq files:
wget http://omixon-download.s3.amazonaws.com/target_brca_example.zip
unzip target_brca_example.zip
```

14.1.2 Customize flow definition, describing the computing cluster

Next, we need to customize the resource requirements based on the computing platform. You may refer to the flow definition format for more details.

```
## customize the resource requirements in flowdef:
- need to change: queue, platform
- may change: walltime, memory, CPUs etc.
vi fastq_bam_bwa.def

## read check flowdef (shell)
flowr as.flowdef x=fastq_bam_bwa.def

## OR from R
as.flowdef(x='fastq_bam_bwa.def')
```

Read and check flowdef

jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	jobid
aln1	scatter	none	none	medium	16384	2:00	12	lsf	1
aln2	scatter	none	none	medium	16384	2:00	12	lsf	2
sampe	scatter	aln1,aln2	serial	medium	16384	2:00	1	lsf	3
fixrg	scatter	sampe	serial	medium	16384	2:00	1	lsf	4
merge	serial	fixrg	gather	medium	16384	12:00	1	lsf	5

A flow definition with default values has already been supplied, briefly,

- Submission Type (**sub_type**): determines, how each step is **submitted** to the cluster. All steps except `merging` may have multiple subprocess (each of which can run in parallel). Thus, we spread (`scatter`) them across the cluster.
- Previous Jobs (**prev_jobs**): The two `aln` steps of `bwa` may be run in parallel, and its subsequent `sampe` would wait for both. Spcecifically, in case of multiple fastq files i^{th} `sampe` step would wait for i^{th} `aln1` and `aln2` steps.
- **cpu_reserved**: Since `aln` can use multiple cores, we provide it 12 cores, and for rest of the steps, 1 core each.
- **walltime**: Merging may take a little longer, so we give it ample amount of time (12 hours). Some computing platforms specify time as `hh:mm:ss` and others prefer `hh:mm`, you may need to check with your system admin.
- **memory**: For simplicity we can assign 16GB (16000kb) of memory to each of these steps (may be an overkill, please change as necessary).
- **queue**: We use a generic `medium` queue, since if usually exists; please change as needed.
- **platform**: Finally, specify the platform of your computing cluster (moab, lsf, torque, sge, slurm [alpha])

Tip: Once we define the flow definition correctly, we may not need to change it any further (one time effort).

14.1.3 Submit to cluster

14.1.3.1 Single step cluster submission

Note: Assuming that the pipeline along with its `.def` and `.conf` files is available in `~/flowr/pipelines`. Also, `.conf` files should have all the correct paths and `.def` file should have resource requirements specified correctly.

```
## get input fastqs
fqs1=~/flowr/genomes/target_brca_example/brca.example.illumina.0.1.fastq
fqs2=~/flowr/genomes/target_brca_example/brca.example.illumina.0.2.fastq

## submit to the cluster
flowr run x=fastq_bam_bwa fqs1=$fqs1 fqs2=$fqs2 samplename=samp execute=TRUE

## change the platform specified in flowdef
flowr run x=fastq_bam_bwa fqs1=$fqs1 fqs2=$fqs2 samplename=samp execute=TRUE platform=slurm
```

```
library(flowr)
fqpath = "~/flowr/genomes/target_brca_example"
## demonstrating that multiple fqs can be used here...
fobj = run(x = "fastq_bam_bwa", samplename = "samp1", execute = TRUE,
        fqs1 = rep(paste0(fqpath, "/brca.example.illumina.0.1.fastq"), 2),
        fqs2 = rep(paste0(fqpath, "/brca.example.illumina.0.2.fastq"), 2))
```

OR from R using:

Refer to the help pages for more details on the run function.

14.1.3.2 Details regarding cluster submission

The **run** function performs several steps, finally submitting the commands to the cluster. It may be useful to go through these steps to understand the details.

1. Get user inputs

Using the name of the pipeline, `run` fetches it in various places inclusing `~/flowr/pipelines`.

```
library(flowr)
setwd("~/flowr/pipelines")
source("fastq_bam_bwa.R")
## this may throw a warning if paths do not exist
## if you have used modules instead of full paths please ignore the warnings
load_opts("fastq_bam_bwa.conf")

## Get example input
## these can be a vector of multiple paired-end files
## OR multiple single-end files
fqs1 = "~/flowr/genomes/target_brca_example/brca.example.illumina.0.1.fastq"
fqs2 = "~/flowr/genomes/target_brca_example/brca.example.illumina.0.2.fastq"
samp = "samplename"

## optionally specify the center, lane, platform etc.
set_opts(rg_center = "the_institute", rg_lane = "1")

## **Note:** load_opts checks if variables ending with
## _exe, _path, _dir etc. exist or not.
## make sure they are all correct.
## Ignore the warnings, if instead of specifying full path to a tool
## you are using the module command.
```

Refer to the help pages of fetch_pipes and fetch_pipes for more details.

2. Read flow definition

```
def = as.flowdef("fastq_bam_bwa.def")
```

```
## def seems to be a file, reading it...
##
## checking if required columns are present...
## checking if resources columns are present...
## checking if dependency column has valid names...
## checking if submission column has valid names...
## checking for missing rows in def...
## checking for extra rows in def...
## checking submission and dependency types...
```

The plot would work only if you have X11 etc enabled, i.e. if you logged into the cluster using `ssh -X` (or `ssh -Y`).

Optionally, one can edit all config files on their own machine, debug and sort issues; when done, upload them to the cluster and submit.

```
plot_flow(def) ## on a cluster, only works if graphics X11 is enabled. ssh -X
```

3. Create a table with all commands to run

We use the function `fastq_bam_bwa` to create a flow mat.

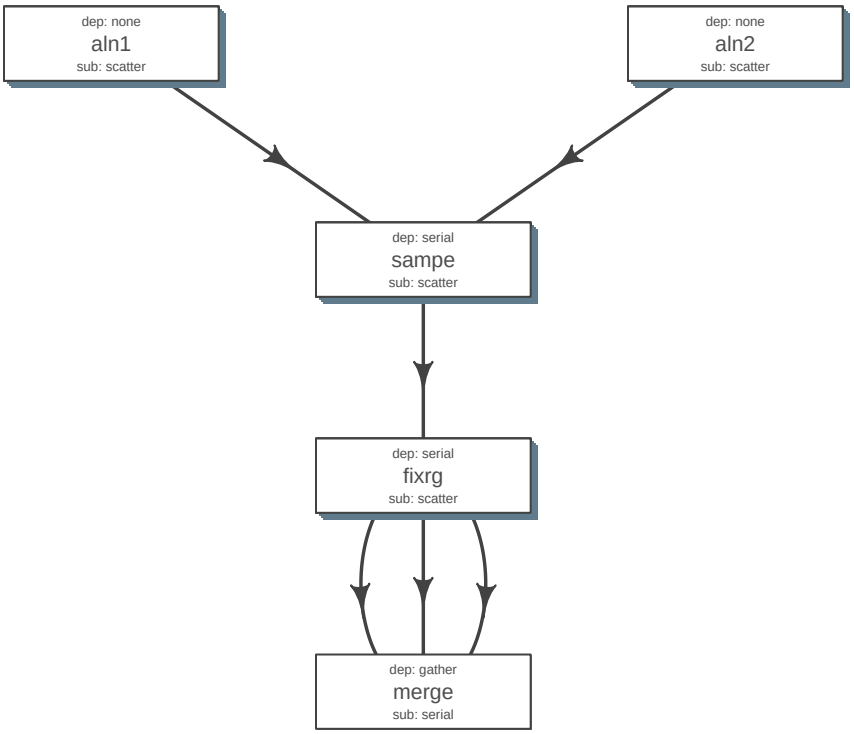

Figure 7:

```
## run the module and create a flow mat, with all the commands
out = fastq_bam_bwa(fqs1, fqs2, samplename = samp)

## optionally, write this to a file (a simple tab delimited table)
write_sheet(out$flowmat, "fastq_bam_bwa.tsv")
```

4. Executing on the computing cluster

Now we can submit this to the cluster using:

```
fobj2 = to_flow(x='~/flowr/pipelines/fastq_bam_bwa.tsv',
                def='~/flowr/pipelines/fastq_bam_bwa.def',
                name = "fastq_bam_bwa",
                execute=TRUE)
```

OR from the terminal using:

```
flowmat=~/flowr/pipelines/fastq_bam_bwa.tsv
flowdef=~/flowr/pipelines/fastq_bam_bwa.def
flowr to_flow x=$flowmat def=$flowdef name=fastq_bam_bwa execute=TRUE
```

Tip: This example shows a single sample, but you may have as many samples in the flowmat. In case of multiple samples, the **samplename** column is used to group commands and each set if submitted as a individual flow.

Several other functions, one may use after submission:

checking the status:

```
## from R:
status(x="~/flowr/runs/fastq_bam_bwa*")

## OR from terminal using:
flowr status x=~/flowr/runs/fastq_bam_bwa*
```

	total	started	completed	exit_status	status
001.aln1	1	1	0	0	processing
002.aln2	1	1	0	0	processing
003.sampe	1	0	0	0	pending
004.fixrg	1	0	0	0	pending
005.merge	1	0	0	0	pending

Additionally, you may **kill** or **rerun** the flow as well.

```
flowr kill x=~/flowr/runs/fastq_bam_bwa*
flowr rerun x=~/flowr/runs/<full path of the flow> start_from=fixrg
```

Please use the respective help pages for more details.