My Report

Contents

flowr: Streamline your workflows

This framework allows you to design and implement complex pipelines, and deploy them on your institution's computing cluster. This has been built keeping in mind the needs of bioinformatics workflows. However, it is easily extendable to any field where a series of steps (shell commands) are to be executed in a (work)flow.

Highlights

- Effectively process a pipeline multi-step pipeline, spawning it across the computing cluster
- **Example:**
  - A typical case with next-generation sequencing, a sample with tens of fastq files)
  - Each file can be processed (aligned) individually, each using multiple cores
  - Say 50 files using 10 cores each, totalling 500 cores across several machines, for one sample
  - flowr further supports processing multiple samples in parrellel, spawning thousands of cores.
- Reproducible, with cleanly structured execution logs
- Track and re-run flows
- Lean and Portable, with easy installation
- Supports multiple platforms (torque, lsf, sge, slurm . . .)

A few lines, to get started:

```
## From the official R repository (may be a few versions behind)
install.packages("flowr")

## OR

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

library(flowr) ## load the library
setup() ## copy flowr bash script; and create a folder flowr under home.
run('sleep', execute=TRUE,  platform='moab') ## submit a simple example
```

- Here is a shiny app, flow_creator which helps you build a flow.
- A few slides providing a quick overview.

Aknowledgements

- Jianhua Zhang
- Samir Amin
- Kadir Akdemir
- Ethan Mao
- Henry Song
- An excellent resource for writing your own R packages: r-pkgs.had.co.nz

Get started

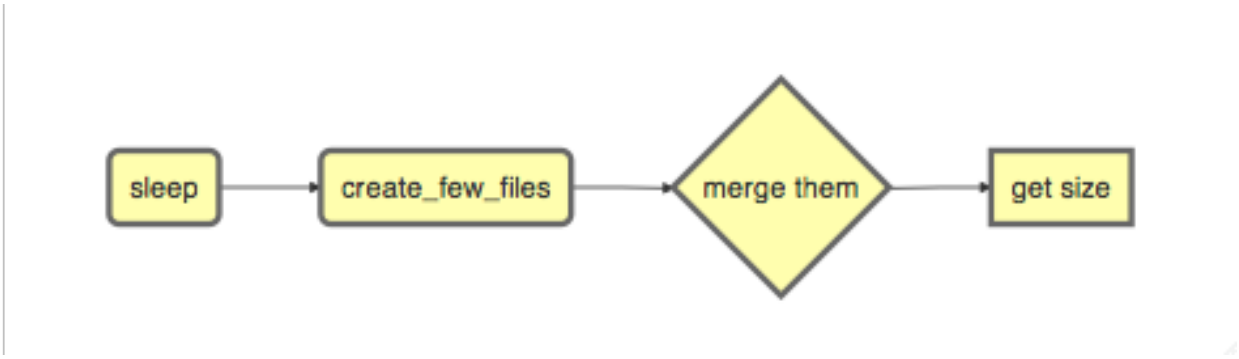
```
install.packages('devtools')
devtools::install_github("sahilseth/flowr")
```

Run a setup function which copies 'flowr' helper script to enable using flow from shell terminal itself. A few examples here.

```
library(flowr)
```

```
setup()
```

Toy example



A simple example where we have three instances of sleep (wait for few seconds), after completion three tmp jobs are started which create files with some random data. After all these are complete, a merge step follows, which combines them into one big file. Next we use `du` to calculate the size of the resulting file.

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.

The table below is referred to as 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

samplename	jobname	cmd
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

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

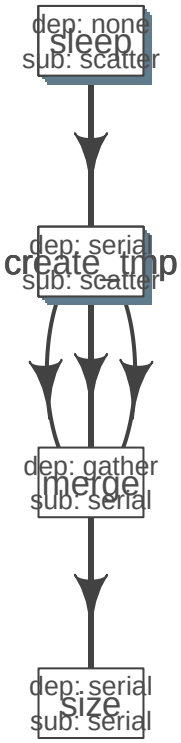
jobname	sub_type	prev_jobs	dep_type	queue	memory_reserved	walltime	cpu_reserved	platform	job
sleep	scatter	none	none	short	2000	1:00	1	torque	
create_tmp	scatter	sleep	serial	short	2000	1:00	1	torque	
merge	serial	create_tmp	gather	short	2000	1:00	1	torque	
size	serial	merge	serial	short	2000	1:00	1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

Stitch it

```
fobj <- to_flow(x = flow_mat, def = as.flowdef(flow_def),
    flowname = "example1", platform = "lsf")
```

Plot it

```
plot_flow(fobj)
```



Test it

Dry run (submit)

```
submit_flow(fobj)
```

```
Test Successful!
You may check this folder for consistency. Also you may re-run submit with execute=TRUE
 ~/flowr/type1-20150520-15-18-27-5mSd32G0
```

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")
OR
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE
```

Check its status

```
flowr status x=~/flowr/type1-20150520-15-18-46-sySOzZnE
```

```
Loading required package: shape
Flowr: streamlining workflows
Showing status of: /rsrch2/iacs/iacs_dep/sseth/flowr/type1-20150520-15-18-46-sySOzZnE


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

Kill it

```
## kill one flow
## flowr kill_flow x=flow_wd

## In case path matches multiple folders, flowr asks before killing
kill(x='fastq_haplotyper*')
##  Flowr: streamlining workflows
##  found multiple wds:
##  ./fastq_haplotyper-MS132-20150825-16-24-04-0Lv1PbpI
##  /fastq_haplotyper-MS132-20150825-17-47-52-5vFIkrMD
```

```
##  Really kill all of them ? kill again with force=TRUE

## submitting again with force=TRUE will kill them:
kill(x='fastq_haplotyper*', force = TRUE)
```

Even if you want to kill the flow, its best to let submit_flow do its job, when done simply use kill(flow_wd). If submit_flow is interrupted, flow detail files etc are not created, thus flowr can't associate submitted jobs with flow instance.

Re-run a flow

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

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

In either case there are two things which are always required, a flow_wd (the folder created by flowr which contains execution logs) and name of the step from where we want to start execution. Refer to the help section for more details.

Interested? Here are some details on building pipelines

Building Pipelines

An easy and quick way to build a workflow is create two separate files. First is a table with commands to run, 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.

Both these files have a `jobname` column which is used as a ID to connect them to each other.

Ingredients for building a pipeline

Essentially there are two main components which go into building a flowr pipeline, a flow matrix with commands to run and a flow definition with details regarding how to stich a pipeline.

Let us read some files to see what they look like and what they do.

```
## ------ load some example data
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"))
```

1. Flow Definition

Each row in this table refers to one step of the pipeline. It describes the resources used by this 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 a typical setup, if one step uses 16 cores these would be blocked and not used during processing of several other steps. Thus resulting in blockage and high cluster load (even when actual CPU usage may be low). Being able to tune them, makes this setup quite efficient.

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.

.. warning:: If these (resource requirements) columns not included in the flow_def, their values should be explicitly defined in the submission template.

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	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	

2. Flow mat: A table with shell commands to run

This is also a tab separated table, with a minimum of three columns as defined below:

- `samplename`: A grouping column. The table is split using this column and each subset is treated as a individual flow. This makes it very easy to process multiple samples using a single submission command.
  - 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 defined above.
- `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.

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

Example:

A —-> B —> C —> D

Consider an example with three steps A, B and C. 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.

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.*

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.*

Relationships

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

Serial: one to one relationship

[scatter] —serial—> [scatter]

A is submitted as scatter, A1 through A10. Further B1, requires A1 to complete; B2 requires A2 and so on, but they need not wait for all of step A jobs to complete. Also B1 through B10 are independent of each other.

To set this up, A and B would have `sub_type` `scatter` and B would have `dep_type` as `serial`. Further, since A is an initial step its `dep_type` and `prev_job` would defined as `none`.

Gather: many to one relationship

[scatter] —gather—> [serial]

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.

Burst: one to many relationship

[serial] —burst—> [scatter]

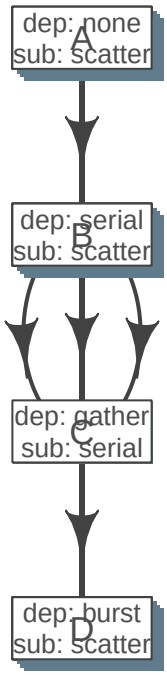
Further, D is a set of three commands (D1-D3), which need to wait for a single process (C1) to complete. They would be submitted as `scatter` after waiting on C in a `burst` type dependency.

In essence and 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"))
fobj = suppressMessages(to_flow(x = ex2mat, def = ex2def))
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(fobj)
```



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

Passing of flow definition resource columns

The resource requirement columns of flow definition are passed along to the final (cluster) submission script.

The following table provides a mapping between the flow definition columns and variables in the submission template (pipelines below).

flow_def_column	hpc_script_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 **: This is gathered from flow_mat

Available Pipelines

Here are some of the available piplines along with their respective locations

name	def	conf	pipe
sleep_pipe	sleep_pipe.def	NA	/Library/Frameworks/R.framework/Versions/
fastq_bam_bwa	NA	NA	/Library/Frameworks/R.framework/Versions/
fastq_bam_rna_ion	NA	NA	/Library/Frameworks/R.framework/Versions/
fastq_bam_variants	NA	NA	/Library/Frameworks/R.framework/Versions/
fastq_haplotyper	fastq_haplotyper.def	fastq_haplotyper.conf	/Library/Frameworks/R.framework/Versions/
fastq_star_rna	NA	NA	/Library/Frameworks/R.framework/Versions/
old_bam_pindel	NA	NA	/Library/Frameworks/R.framework/Versions/
old_bam_preprocess	NA	NA	/Library/Frameworks/R.framework/Versions/
old_bam_xenome	NA	NA	/Library/Frameworks/R.framework/Versions/
old_bwa_pipe	NA	NA	/Library/Frameworks/R.framework/Versions/
old_dna_qc	NA	NA	/Library/Frameworks/R.framework/Versions/
old_fastq_bam_bwa2	NA	NA	/Library/Frameworks/R.framework/Versions/
old_fastq_bismark_meth	NA	NA	/Library/Frameworks/R.framework/Versions/
old_flow_bam_preprocess	NA	NA	/Library/Frameworks/R.framework/Versions/
old_proc_bwa_pipe	NA	NA	/Library/Frameworks/R.framework/Versions/
split_aln_merge	split_aln_merge.def	NA	/Library/Frameworks/R.framework/Versions/
build	NA	NA	/Users/sahilseth/Dropbox2/Dropbox/public/

Cluster Support

Support for several popular cluster platforms are built-in. There is a template, for each platform, which should would 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

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

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 accomodate this one can use regular expressions as described in the relevent section of the flowr config.

For example LSF may show a string such as:

```
Job <335508> is submitted to queue <transfer>.

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 !

3. **render dependency**: After collecting job ids from previous jobs, flowr render 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

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

Platform	command	status	queue.type
LSF 7	bsub	Not tested	lsf
LSF 9.1	bsub	Yes	lsf
Torque	qsub	Yes	torque
SGE	qsub	Beta	sge
SLURM	sbatch	under-dev	slurm

*queue short-name used in flow

- 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

Example of building a pipeline

A pipeline consists of several pieces, namely, a function which generates a flowmat, a flowdef and optionally a text file with parameters and paths to tools used as part of the pipeline.

A R function which creates a flow mat, is a module. Further a module with a flow definition is a pipeline.

We beleive pipeline and modules may be interchangeble, in the sense that a *smaller* pipeline may be included as part of a larger pipeline. In flowr a module OR pipeline always returns a flowmat. The only differnce being, a pipeline also has a correspomding flow definition file. As such, creating a flow definition for a module enables flowr to run it, hence a module **elevates**, becoming a pipeline. This lets the user mix and match several modules/pipelines to create a customized larger pipeline(s).

Let us follow through an example, providing more details regarding this process. Here are a few examples of modules, three functions `sleep`, `create_tmp` and `merge_size` each returning a flowmat.

Define modules

```
#' @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))
}
```

We then define another function `sleep_pipe` which calls the above defined **modules**; fetches flowmat from each, creating a larger flowmat. This time we will define a flowdef for the `sleep_pipe` function, elevating its status from module to a pipeline.

Define the pipeline

```
#' @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))
}
```

Generate a flowmat

Here is how the generated flowmat looks like.

```
out = sleep_pipe(x = 3, "sample1")
flowmat = out$flowmat
```

samplename	jobname	cmd
sample1	sleep	sleep 2 && sleep 1;echo 'hello'
sample1	sleep	sleep 8 && sleep 13;echo 'hello'
sample1	sleep	sleep 1 && sleep 3;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

Create flow definition

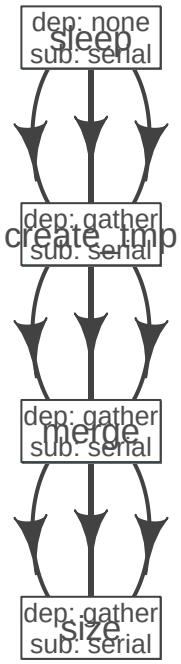
flowr enables us to quickly create a skeleton flow definition using a flowmat, which we can then alter to suit our needs. A handy function to_flowdef, accepts a flowmat and creates a flow definition. The default skeleton takes a very conservative approach, creating all submissions as `serial` and all dependencies as `gather`. This ensures robustness, compromising efficiency. Thus we will enable parallel process where possible, making this into a better pipeline.

Here is how it looks presently:

```
def = to_flowdef(flowmat)
```

```
## Creating a skeleton flow definition
## Following jobnames detected: sleep create_tmp merge size
```

```
plot_flow(suppressMessages(to_flow(flowmat, def)))
```



After making the desired changes, the new pipeline looks better. Alternatively, one may write this to a file and make other desired changes in resource requirements.

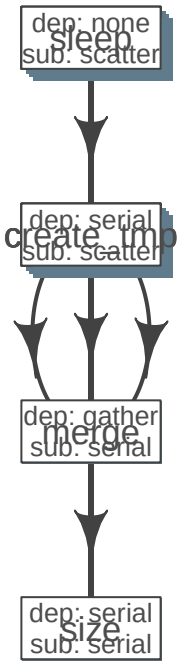
Pipeline follows the following steps, with dependencies mentioned in ():

- multiple sleep commands would run in parallel (none, first step)
- For each sleep, create_tmp creates a tmp file (serial)
- All tmp files are merged; when all are complete (gather)
- Then we get size on the resulting file (serial)

```
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	

```
plot_flow(suppressMessages(to_flow(flowmat, def)))
```


dep: none
sleep
sub: scatter
dep: serial
create_tmp
sub: scatter
dep: gather
merge
sub: serial
dep: serial
size
sub: serial