Cluster Flow

The Manual

Babraham Bioinformatics

Table of Contents

Introduction

What is Cluster Flow?

Cluster Flow is simple package to run pipelines in a cluster environment. It is comprised of several layers:

- cf
  - The main cluster flow command. This is called to initiate a new pipeline run.
- Pipelines
  - Protocols that describe a series of modules to be run, along with any parameters
- Modules
  - The instructions for an individual task. These can be written in any language but must conform to a common API, described within this document.
- Runs
  - Created from the pipeline template for each file. Specifies configuration variables and traces output filenames.

Cluster Flow will set off multiple queued jobs on the cluster with queue dependencies as defined in the pipeline.

How does Cluster Flow work?

A typical Cluster Flow run will work as follows:

- Pipelines and modules are written for Cluster Flow
- The `cf` command is run, initiating a pipeline with a set of files
- CF decides on a number of runs, each with a set of input files or URLs
- Input filenames are checked to make sure that they all have the same file extension
- If input files are fastq, CF tries to work out whether they're paired end or single end from the filenames. It dies with an error if it finds a mixture
- CF checks that the files exist (unless they're URLs) and parses the pipeline file
- A run file is created for each run. This contains the configuration variables, a copy of the pipeline used and the starting filenames with the associated id `start_000`
- CF submits all of the jobs to the cluster
  - Each module in each run has its own cluster job
  - Modules are submitted with dependencies so that they execute in the order specified by the pipeline
- Main CF program finishes

- Modules execute, using the run file and the previous job ID to find their input files
- STDERR is appended to the log files.
    - Commands should print their commands with `###CFCMD` prepended
    - Important messages should start with `###CF`
- Upon the completion of the module, the resulting output files are appended to the run file, along with the job ID so that the next module can find the files
- Once all pipeline modules have finished, core cluster flow notification modules run
    - These clean up the output file and send e-mails

Using Cluster Flow for the first time

`module load clusterflow`

Cluster Flow is typically installed as an environment module. Before you can use it, you'll need to run `module load clusterflow`. You can add this line to your `.bashrc` script to make it happen automatically each time you log in. To do this you can copy and paste the following command:

```
sed -i "$ a\module load clusterflow" ~/.bashrc
```

`cf --make_config`

Before you start using Cluster Flow, it's a good idea to set yourself up with a personalised Cluster Flow config file. Cluster Flow has an interactive mode which will guide you through the required values and create a config file for you. To run this, run `cf --make_config`

`cf --add_genome`

Most pipelines need a reference genome. Running `--add_genome` gives an interactive wizard which will lead you through the process of adding new genome paths.

Typical usage

To run cluster flow, use the `cf` command.

Most pipelines will need a genome name to be specified. IDs such as NCBIM37 or GRCh37 are used as keys to identify genome paths for aligning reads. These are specified by adding the flag `--genome` followed by an ID when you run Cluster Flow.

After flags are specified, the name of the module or pipeline to be run should be declared. If there is a pipeline with the same name as a module it will take preference.

Finally, write the input filenames. This can be done with linux wildcard expansion (`*`).

For example, to run the `sra_bismark` pipeline, the command would be:

```
cf --genome NCBIM37 sra_bismark *.sra
```

Running Cluster Flow

Command line parameters

Flag	Description
`--genome <ID>`	ID of a genome referred to in `clusterflow.config`
`--genome_path <path>`	Path to a genome to be used for alignment
`--bowtie_path <path>`	Path to a bowtie index base to be used for alignment
`--gtf_path <path>`	Path to a GTF file to be used for alignment (eg. for Tophat)
`--paired`	Force paired-end mode
`--single`	Force single-end mode
`--no_fn_check`	Disable input file type checking
`--file_list`	Text file containing input files or download URLs
`--params`	Specify extra module parameters for this run
`--split_files <num>`	Create one run per <num> files
`--max_runs <num>`	Divide input files into <num> runs. Set as 0 to disable.
`--email <email>`	Set the e-mail address for notifications
`--priority <num>`	Set the queue priority for cluster jobs
`--cores <num>`	Set the maximum number of cores to use for all runs
`--mem <string>`	Set the maximum memory to use for all runs
`--notifications [cresa]`	Specify desired notifications
`--list_pipelines`	Print available pipelines
`--list_modules`	Print available modules
`--list_genomes`	Print available genomes
`--dry_run`	Prints jobs to terminal instead of submitting them to the cluster
`--qstat`	Displays formatted qstat output of your jobs
`--qstatall`	Displays formatted qstat output of all jobs

`--qstatcols`	Colours output from `--qstat` or `--qstatall`
`--qdel <id>`	Delete all jobs from a running pipeline. `<id>` is printed with `--qstat`
`--make_config`	Interactive prompt to generate a personalised CF config file
`--add_genome`	Interactive wizard to add new genomes to your `genomes.config` files
`--version`	Print version of Cluster Flow installed
`--check_updates`	Look for available Cluster Flow updates
`--help`	Print help message

`--genome` Default: none

Some pipelines which carry out a reference genome alignment require a genome directory path to be set. Requirements for format may vary between modules.

`--paired` Default: Auto-detect

If specified, Cluster Flow will send two files to each run, assuming that the order that the file list is supplied in corresponds to two read files. If an odd number of files is supplied, the final file is submitted as single end.

`--single` Default: Auto-detect

If specified, Cluster Flow will ignore its auto-detection of paired end input files and force the single end processing of each input file.

`--no_fn_check` Default: none

Cluster Flow will make sure that all of the input files have the same file extension to avoid accidentally submitting files that aren't part of the run. Specifying this parameter disables this check.

`--file_list` Default: none

If specified, you can define a file containing a list of filenames to pass to the pipeline (one per line). This is particularly useful when supplying a list of download URLs.

`--params` Default: none

Pipelines and their modules are configured to run with sensible defaults. Some modules accept parameters which change their behaviour. Typically, these are set within a pipeline config file. By using `--params`, you can add extra parameters at run time. These will be set for every module in the pipeline (though they probably won't all recognise them).

`--split_files` Default: (config file - typically 1)

Cluster Flow generates multiple parallel runs for the supplied input files when run. This is typically a good thing, the cluster is designed to run jobs in parallel. Some jobs may involve many small tasks with a large number of input files however, and 1:1 parallelisation may not be practical. In such cases, the number of input files to assign to each run can be set this flag.

`--max_runs` Default: none

It can sometimes be a pain to count the number of input files and work out a sensible number to use with `--split_files`. Cluster Flow can take the `--max_runs` value and divide the input files into this number of runs, setting `--split_files` automatically.

A default can be set for `max_runs` in the `clusterflow.config` file, and this value is set to 12 if no value is found in the config files. Set to `0` to disable.

This parameter will override anything set using `--split_files`.

`--email` Default: (config file)

Cluster Flow can send notification e-mails regarding the status of runs. Typically, e-mail address should be set using a personalised `~/clusterflow.config` value (see below). This parameter allows you to override that setting on a one-off basis.

`--priority` Default: (config file - typically -500)

GRID Engine uses a priority system to manage jobs in the queue. Priorities can be set ranging from -1000 to 0.

`--cores` Default: (config file - typically 64)

Override the maximum number of cores allowed for each Cluster Flow pipeline, typically set in the Cluster Flow config file. For more information see ***Avoiding cluster overload***.

`--mem` Default: (config file - typically 128G)

Setting `–mem` allows you to override the maximum amount of simultaneously assigned memory. For more information see ***Avoiding cluster overload***.

`--notifications` Default: (config file - typically cea)

Cluster Flow can e-mail you notifications about the progress of your runs. There are several levels of notification that you can choose using this flag. They are:

- `c` Send notification when all runs in a pipeline are completed
- `r` Send a notification when each run is completed
- `e` Send a notification when a cluster job ends
- `s` Send a notification if a cluster job is suspended
- `a` Send a notification if a cluster job is aborted

Setting these options at run time with the `--notifications` flag will override the settings present in your `clusterflow.config` configuration files.

Note – setting the `s` flag when using many input files with a long pipeline may cause your inbox to be flooded.

`--qstat`, `--qstatall` and `--qstatcols`

When you have a lot of jobs running and queued, the qstat summary can get a little overwhelming. To combat this and show job hierarchy in an intuitive manner, you can enter into the console `cf --qstat`. This parses qstat output and displays it nicely. `cf --qstatall` does the same but for all jobs by all users.

To make the output easier to read, you can use the additional parameter `--qstatcols`. How this will look depends entirely on your terminal colour setup. It works nicely in terminal windows with light backgrounds and can look really horrible on terminal windows with dark backgrounds. For this reason it is only enabled when specified.

To make the command a little less clumsy, you can create aliases in your `.bashrc` script. Typically, these two lines:

```
alias qs='cf --qstat --qstatcols'
alias qsa='cf --qstatall --qstatcols'
```

If you're feeling lazy, you can append these lines to your `.bashrc` script through the command line by copying and pasting the following commands:

```
sed -i "$ a\alias qs='cf --qstat --qstatcols'" ~/.bashrc
sed -i "$ a\alias qsa='cf --qstatall --qstatcols'" ~/.bashrc
```

`--qdel`

Sometimes you may be running multiple pipelines and want to stop just one. It can be a pain to find the job numbers to do this manually, so instead you can use Cluster Flow to kill these jobs. When running `cf --qstat`, ID values are printed for each pipeline. Use this with `--qdel`. eg:

```
cf --qdel sra_bowtie_1391074179
```

Paired end / single end files

If using Cluster Flow with FastQ files, it will try to guess whether the files are paired end or single end. This is done by sorting the filenames, stripping `_[1-4]` and then comparing each file name to the next. If a pair is found to be identical, they are processed as paired end files.

When input files aren't FastQ files, each Cluster Flow module will try to determine paird end or single end data by the same method. This behaviour is particularly useful when processing SRA files, as a single input file can split into multiple paired end files. Downstream modules can be

run blindly and will behave correctly according to the output `.fastq` files produced by `fastq_dump`.

This behaviour can be overridden by specifying `--paired` or `--single` on the command line.

Downloading files

When downloading files from an FTP server, parallelisation can be a problem. Cluster Flow has a download function built into its core `cf` package to deal with downloads. Cluster Flow sets off each download as a cluster job, with queue IDs set so that they process in series. The pipeline jobs also wait on the download IDs, so as soon as a download is finished it begins processing. In this way, processing does not need to wait for all downloads to finish, yet downloads are able to run in series and not overwhelm the server or internet connection.

To use this feature, simply submit download URLs instead of input filenames. Cluster Flow will recognise anything starting with `http`, `https` or `ftp` as a URL and set it off with the `download` module.

If using the `--file-list` parameter you can specify a filename to save each download. Add this on the same line as the download URL, separated by a tab character. This is particularly useful when downloading arbitrarily named SRA files. Labrador is able to generate these download files, offering a simple way to get download URLs and sensibly named files.

An example download `--file-list` file is below:

```
ftp://ftp-trace.ncbi.nlm.nih.gov/sra/SRR944695.sra	Input_4OHT_rep5.sra
ftp://ftp-trace.ncbi.nlm.nih.gov/sra/SRR944694.sra	Input_4OHT_rep4.sra
ftp://ftp-trace.ncbi.nlm.nih.gov/sra/SRR944693.sra	Input_4OHT_rep3.sra
ftp://ftp-trace.ncbi.nlm.nih.gov/sra/SRR944692.sra	Input_4OHT_rep2.sra
ftp://ftp-trace.ncbi.nlm.nih.gov/sra/SRR944691.sra	Input_4OHT_rep1.sra
```

Avoiding cluster overload

If using Cluster Flow with a large number of files it can be easy to swamp the available resources on the cluster and annoy any other users trying to get things done. Cluster Flow has several built in features to try to avoid this.

Firstly, CF can limit the number of parallel runs by using either `--split_files` or `--max_runs`. By default, Cluster Flow runs with a maximum of 12 parallel runs per pipeline. This default can be modified in the Cluster Flow config file or at run time.

In addition to limiting the number of parallel jobs it runs, Cluster Flow can try to limit the memory usage and number of cores each module uses. It calculates the maximum number of jobs that can be theoretically running at the same time when a pipeline is initiated. The available cores and memory are divided by this number and these ideal cores and memory per

module are presented these numbers to each module. The modules can compare these values to their minimum and maximum requirements and request appropriate resources.

The end result of this process should be that jobs are run with maximum resources and speed when there are not too many files to process. Jobs will be run with minimum resources when processing many files so as not to overwhelm the available compute resources.

This behaviour is configured using the `@total_cores` and `@total_mem` configuration options, which can be overridden with the `--cores` and `--mem` command line arguments.

Configuring Cluster Flow

Config file wizard

Cluster Flow comes with a command line wizard which can walk you through the creation of your personalised config file.

Simply run `cf --make_config` and answer the on screen questions and cluster flow will generate ~/clusterflow.config for you. Sensible defaults are suggested.

clusterflow.config

Cluster flow will search three locations for a config file every time it is run. Variables found in each file can override those read from a previous config file. They are, in order of priority:

- `<working directory>/clusterflow.config`
  - A config file found in the current working directory when a pipeline is executed has top priority, trumped only by command line parameters
- `~/clusterflow.config`
  - A config file in your home directory can be used to set parameters such as notification level and e-mail address
- `<installation directory>/clusterflow.config`
  - A config file in the Cluster Flow installation directory is ideal for common settings specific to the environment

Config files contain key: value pairs. Syntax is as follows: `@key value` (tab delimited, one per line). Cluster Flow ships with an example config file called `clusterflow.config.example`

Basic settings

@email

Sets your e-mail address, used for e-mail notifications.

@check_updates

Cluster Flow can automatically check for new versions. If an update is available, it will print a notification each time you run a job. You can specify how often Cluster Flow should check for updates with this parameter. The syntax is a number followed by `d`, `w`, `m` or `y` for days, weeks, months or years. Cluster Flow will check for an update at runtime if this period or more has

elapsed since you last ran it. You can disable update checks and alerts by setting `@check_updates 0` in your `~/clusterflow.config` file.

You can get Cluster Flow to manually check for updates by running `cf --check_updates`

@split_files

The default number of input files to send to each run. Typically set to 1.

@priority

The priority to give to cluster jobs. Default is -500 to avoid swamping the cluster for all other users.

@max_runs

The maximum number of parallel runs that cluster flow will set off in one go. Default is 12 to avoid swamping the cluster for all other users.

@notification

Multiple `@notification` key pairs can be set with the following values:

- complete
  - An e-mail notification is sent when all processing for all files has finished
- run
  - An e-mail is sent when each run finishes (each set of input files)
- end
  - A `qsub` notification e-mail is sent when each cluster job ends. Likely to result in a full inbox!
- suspend
  - A `qsub` notification e-mail is sent if a job is suspended
- abort
  - A `qsub` notification e-mail is sent if a job is aborted

Cluster Flow sends the `run` and `complete` notifications using the `cf_run_finished` and `cf_runs_all_finished` modules. These modules handle several tasks, such as cleaning useless warning messages from log files. E-mails contain the contents of all log files, plus a section at the top of highlighted messages, specified within log messages by being prefixed with `###CF`.

@total_cores

The total number of cores available to a Cluster Flow pipeline. Modules are given a recommended number of cores so that resources can be allocated without swamping the cluster.

@total_mem

The total amount of memory available to a Cluster Flow pipeline. Modules are given a recommended quota so that resources can be allocated without swamping the cluster.

@ignore_modules

If you do not use environment modules on your system, you can prevent Cluster Flow from trying to use them (and giving a warning) by adding this line to your config file.

@cluster_environment

Cluster Flow can submit jobs to both GRIDEngine and LSF cluster management systems. This configuration variable sets which environment to use. The possible options are:

- GRIDEngine
  - The default setting. This is used if this setting is absent or not recognised.
- LSF

It is worth noting that Cluster Flow has been developed on a GRIDEngine Cluster so is likely to be most robust on similar setups. At the time of writing, the `--qstat` and `--qstatall` parameters only work with the GRIDEngine environment.

Genome paths

Many modules within Cluster Flow require reference genomes for alignment. Cluster Flow comes with an interactive wizard to help you add new genome paths, just run: `cf --add_genome`

Genome paths are stored within files called `genomes.config` which are stored in the same directories as `clusterflow.config`. Within this file, each path is described with `@genome_path` followed by a key used when submitting the Cluster Flow run (eg. `--genome GRCh37`) followed by an absolute path. Optionally, species and assembly can be added after this.

There are three types of paths that can be specified:

- @genome_path
  - The directory containing a reference genome
- @bowtie_path
  - The file name base of a set of bowtie indices
- @gtf_path
  - The file name of a GTF file for a given genome.

All three types of path should share genome keys if applicable. The fields should be separated by a tab character. Cluster Flow ships with an example genomes file called `genomes.config.example`

Writing Pipelines

Pipeline syntax

All pipelines conform to a standard syntax. The name of the pipeline is given by the filename, which should end in `.config`. The top of the file should contain a title and description surrounded by `/*` and `*/`

Variables are set using the same `@key value` syntax as in clusterflow.config files. Typical variables for pipelines are `@require_genome`, `@require_bowtie` or `@require_gtf`

Modules are described using # prefixes. Tab indentation denotes dependencies between modules. Syntax is `#module_name parameters`, where there can be any number of space separated parameters which will be passed on to the module at run time.

Example pipeline

Here is an example pipeline, which requires a genome path and uses three modules:

```
/*
Example Pipeline
----------------
This pipeline is an example of running three modules which depend on each other.
Module 2 is run with a parameter that modifies its behaviour. This block of text
is used when cf --help example_pipeline is run
*/

#module1
	#module2
	#module2 parameter
		#module3
```

Troubleshooting

- Try running `dos2unix` on your pipeline.config file

Writing Modules

Module syntax

Modules do not conform to any syntax *per se* – they can be written in any language and in any style, though they must conform to a common API so that cluster flow can interact with them.

Required command line flags

All modules will be called directly by Cluster Flow before any cluster jobs are set up, to determine their required job parameters. Each module must return a value for the following command line flags:

Flag	Description
`--cores <num>`	Print required number of cores
`--mem <num>`	Print required amount of memory
`--modules`	Print names of required environment modules
`--help`	Print Help

`--cores`

Cluster Flow will calculate how many cores are available for the module before launching it as a job on the cluster. It will call the module directly with the `--cores` flag and the suggested number of cores that the module can use. The module should print a single integer representing the number of cores that it requires and exit.

The maximum number of cores is a recommendation only. Cluster Flow will assign whatever the module returns.

`--mem`

Much like cores, Cluster Flow will calculate the maximum memory available for the module and call the module with the `--mem` flag followed by an amount of memory in bytes. The module should print the required memory and exit. Required memory can be returned in bytes or formatted with a number followed by a single letter suffix; *eg.* `4G`, `4096M` or `4294967296`.

You can use the core Cluster Flow helper functions `allocate_cores()` and `allocate_memory()` to convert simple human readable strings to bytes. For example:

```
return CF::Helpers:: allocate_memory($allocated, '2G', '10G');
```

The maximum available memory is a recommendation only. Cluster Flow will assign whatever the module returns.

`--modules`

Most Cluster Flow modules will require a system program to exist in the PATH. It's common practice to use environment modules to manage this. Environment modules need to be loaded from the head node before the `qsub` jobs are set off. Cluster Flow calls each CF module with the `--modules` flag before running it. A comma separated list of module names should be printed. Each of these will be loaded with `module load <name>`.

`--help`

Cluster Flow can be called with the `--help` flag followed by any pipeline or module name. If a module name is requested, Cluster Flow will call that module with the `--help` parameter and print the STDOUT.

Command line parameters

When a module is called by its cluster job, it will have the following parameters passed in this order:

- Run file filename
- Cluster job ID
- Previous job ID
- Number of cores assigned
- Amount of memory assigned
- Extra parameters

A run file is created by Cluster Flow for each batch of files. It describes variables to be used, the pipeline specified and the filenames used by each module. The syntax of variables and pipeline is described in Pipeline syntax.

File names are described by a job identifier followed by a tab then a filename. Each module is provided with its own job ID and the ID of the job that was run previously. By using these identifiers, the module can read which input files to use and write out the resulting filenames to the run file when complete. Example run file syntax:

```
first_job_938       filename_1.txt
first_job_938       filename_2.txt
second_job_375      filename_1_processed.txt
second_job_375      filename_2_processed.txt
```

There can be any number of extra parameters, these are specific the module and are specified in the pipeline configuration.

E-mail report highlights

Any STDOUT or STDERR that your module produces will be written to the Cluster Flow log file. At the end of each run and pipeline, an e-mail will be sent to the submitter with details of the run results (if specified by the config settings). Because the log file can be very long Cluster Flow pulls out any lines starting with `###CF`. Typically, such a line should be printed when a module finishes, with a concise summary of whether it worked or not.

Modules should print the command that they are going to run to STDERR so that this is recorded in the log file. These are also sent in the e-mail notification and should start with `###CFCMD`

Exit codes

GRID Engine will continue to fire off dependent jobs as soon as the parent jobs finish, irrespective of their output. If a module fails, the cleanest way to exit is with a success code, but without printing any resulting output filename. The following modules will not find their input filenames and so should immediately exit.

Example module

An example module is distributed with Cluster Flow in `modules/example_module`

The module doesn't do anything but shows the typical workflow for a Perl module, replete with lots of comments saying what everything is doing. Typically, the easiest way to write a new module is to copy and old one, and modify it to suit your needs.

If you write a new module, please let us know! We'd love to package it with Cluster Flow so that other people can benefit.

Troubleshooting

- Try running `dos2unix` on your module script
- Make sure that your module permissions allow it to be executed
    - `chmod 0775 module_name`
- Make sure that your module permissions are still good after exporting from a subversion directory, if you are using one
    - `svn propset svn:executable true modules/*`

Cluster Flow Perl Modules

If your module is written in Perl, there are some common Cluster Flow packages that you can use to provide some pre-written functions.

Using Cluster Flow packages

There are currently three packages available to Cluster Flow modules. `Helpers` contains subroutines of general use for most modules. `Constants` and `Headnodehelpers` contain subroutines primarily for use in the main `cf` binary. You can include the Helpers package by adding the following to the top of your module file:

```
use FindBin qw($Bin);
use lib "$FindBin::Bin/../source";
use CF::Helpers;
```

We use the package `FindBin` to add the binary directory to the path (where `cf` is executing from).

Cluster Flow helper functions

`load_runfile_params(@ARGV);`

This function reads the run file and returns a list of input filenames. Note that `$files` is a referenced array and `$config` is a referenced hash.

```
my ($files, $runfile, $job_id, $prev_job_id, $cores, $mem, $parameters, $config) =
CF::Helpers::load_runfile_params(@ARGV);
```

`is_paired_end(@files);`

This function takes an array of file names and returns an array of single end files and an array of arrays of paired end files. It sorts the files alphabetically, removes any occurance of `_[1-4]` from the filename and compares. Identical pairs are returned as paired end. This function will return the appropriate file arrays if `--paired` or `--single` was used at run time.

```
my ($se_files, $pe_files) = CF::Helpers::is_paired_end(@$files);
foreach my $file (@$se_files){
	print “$file is single end\n”;
}
foreach my $files_ref (@$pe_files){
	my @files = @$files_ref;
	print $files[0].” and “.$files[1].” are paired end.\n”;
}
```

is_bam_paired_end($files);

Looks at BAM/SAM file headers and tries to determine whether it has been generated using paired end input files or single end. The subroutine simply searches the `@PG` header for the occurrence of `-1` and `-2`.

```
if(CF::Helpers::is_bam_paired_end($file)){
	# do something with paired end BAM
} else {
	# do something with single end BAM
}
```

fastq_encoding_type($file);

Scans a FastQ file and tries to determine the encoding type. Returns strings `integer`, `solexa`, `phred33`, `phred64` or `0` if too few reads to safely determine. This is done by observing the minimum and maximum quality scores.

For more details, see http://en.wikipedia.org/wiki/FASTQ_format#Encoding

```
($encoding) = CF::Helpers::fastq_encoding_type($file);
```

fastq_min_length($file);

Scans the first 100000 reads of a FastQ file and returns the longest read length that it finds.

parse_seconds($raw, [$long]);

Simple function that takes time in seconds as an input and returns a human readable string. The optional second `$long` variable determines whether to use h/m/s (`0`, false) or hours/minutes/seconds (`1`, true) and defaults to true.

human_readable_to_bytes($raw);
bytes_to_human_readable($raw);

Two functions which convert between human readable memory strings (eg. `4G` or `100M`) and bytes.

allocate_cores($recommended, $min, $max);

Takes the suggested number of cores to use, a minimum and maximum number and returns a sensible result.

allocate_memory($recommended, $min, $max);

Takes the suggested number of memory to use, a minimum and maximum amount and returns a sensible result. Input can be human readable strings or bytes. Returns a value in bytes.

Getting Help

If you're struggling with Cluster Flow, please do get in touch.

Updates will be posted on our website - http://www.bioinformatics.babraham.ac.uk/projects/clusterflow/

You can get in touch with us at babraham.bioinformatics@babraham.ac.uk

You can get in touch with the author of this tool at phil.ewels@babraham.ac.uk (his personal site is at http://phil.ewels.co.uk)