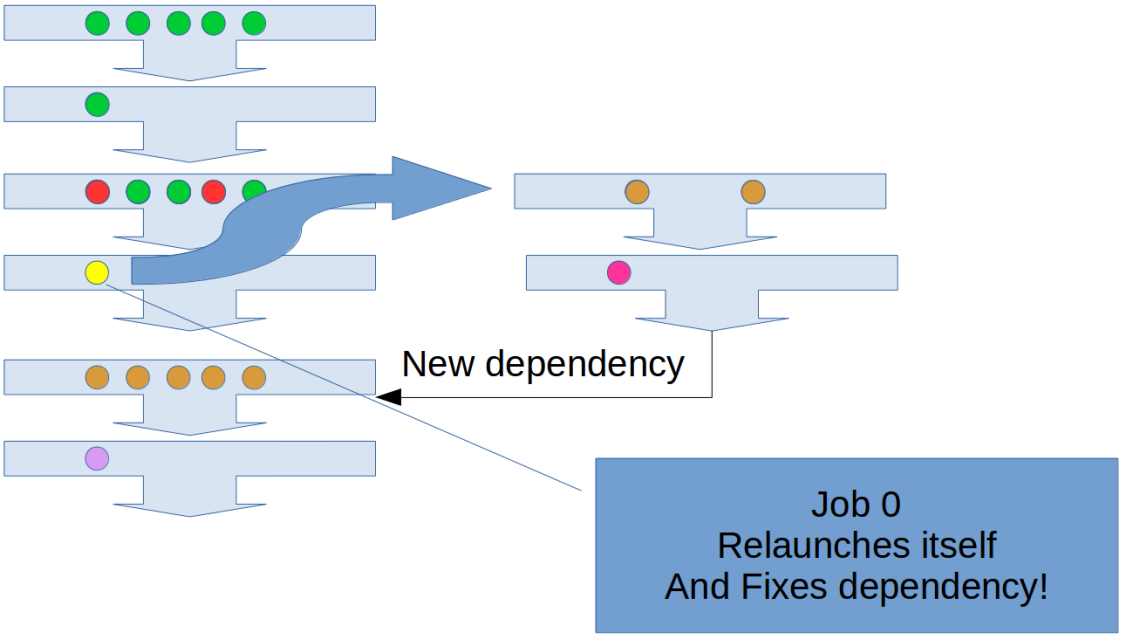
# Decimate Documentation

***Release 0.5***

**Samuel KORTAS**

**Feb 19, 2018**

# User Documentation


New dependency
Job 0
Relaunches itself
And Fixes dependency!

CHAPTER 1

# What is *Decimate*?

Developped by the KAUST Supercomputing Laboratory (KSL), *Decimate* is a SLURM extension written in Python allowing the user to handle jobs per hundreds in an efficient and transparent way. In this context, the constraint limiting the number of jobs per users is completely masked. The time consuming burden of managing thousands of jobs by hand is also alleviated by making available to the user the concept of workflow gathering a set of jobs that he can manipulate as a whole.

*Decimate* is released as an Open Source Software under BSD Licence. It is available at

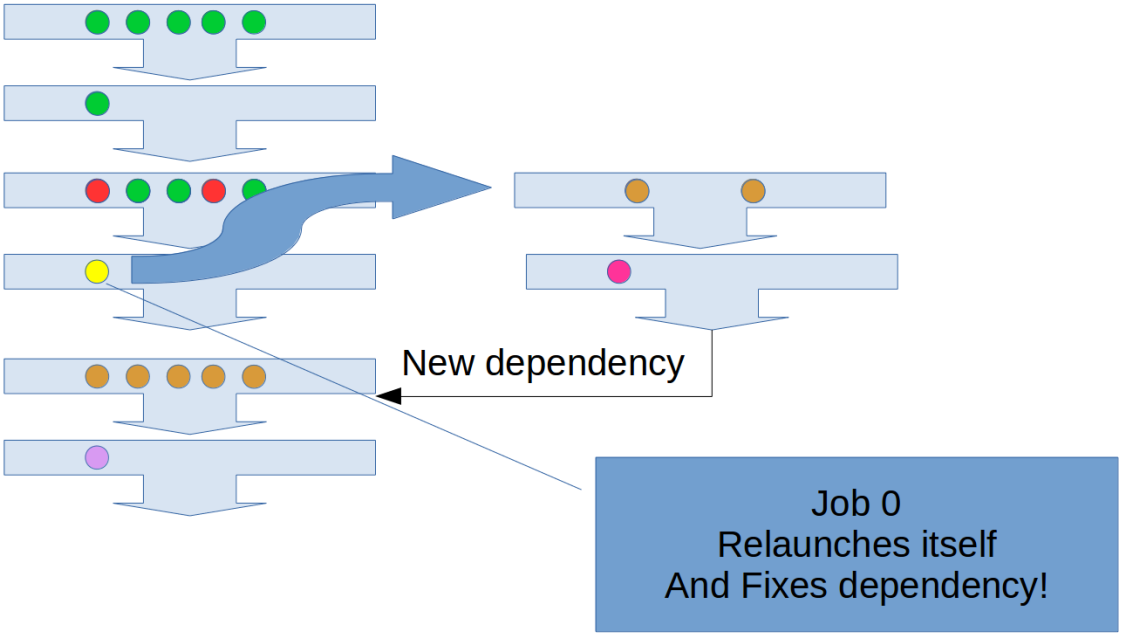
## 1.1 Features

*Decimate* allows a user to:

- Submit any number of jobs regardless of any limitation set in the scheduling policy on the maximum number of jobs authorized per user.
- Manage all the submitted jobs as a single workflow easing their submission, monitoring, deletion or reconfiguration.
- Ease the definition, submission and management of jobs run on a large set of combinations of parameters.
- Benefit from a centralized log file, a unique point of capture of relevant information about the behavior of the workflow. From Python or shell, at any time and from any jobs, the logging levels info, debug, console and mail are available.
- Send fully-configurable mail messages detailing the current completion of the workflow at any step of its execution.
- Easily define a procedure (in shell or Python) to check for correctness of the results obtained at the end of given step. Having access to the complete current status of the workflow, this procedure can make the decision on-the-fly either to stop the whole workflow, to resubmit partially the failing components as is, or to modify it dynamically.

## 1.2 Automated restart in case of failure

In case of failure of one part of the workflow, *Decimate* automatically detects the failure, signals it to the user and launches the misbehaving part after having fixed the job dependency. By default if the same failure happens three consecutive times, *Decimate* cancels the whole workfow removing all the depending jobs from the scheduling. In a next version, *Decimate* will allow the automatic restarting of the workflow once the problem causing its failure has been cured.




## 1.3 Fully user configurable environment

*Decimate* also allows the user to define his own mail alerts that can be sent at any point of the workflow.

Some customized checking functions can also be designed by the user. Their purpose is to validate if a step of the workflow was succesful or not. It could involved checking for the presence of some result files, grepping some error or success messages in them, computing ratio or checksum... These intermediate results can be easely transmitted to *Decimate* validating or not the correctness of any step. They can also be forwarded by mail to the user where as the workflow is executing.

CHAPTER 2

# Using *Decimate*

Via *Decimate*, four commands are added to the user environment: **dbatch** to submit workflows, **dstat** to monitor their current status, **dlog** to tail the log information produced and **dkill** to cancel the execution of the workflow.

## 2.1 Supported Workflows

For *Decimate*, a *workflow* is a set of jobs submitted from a same directory. These jobs can depend on one another and be job array of any size.

How job are named: *job_name-attempt-array*

## 2.2 Submitting a job

### 2.2.1 options

*Decimate* **dbatch** command accepts the same options as the SLURM **sbatch** command and extends it in two ways:

- it transparently submits the user job within a fauit-tolerant framework
- it adds new options to manage the workflow execution if a problem occurs
  - `--check=SCRIPT_FILE` points to a user script (either in python or shell) to validate the correctness of the job at the end of its execution
  - `--max-retry=MAX_RETRY` setting number of time a step can fail and be restarted automatically before failing the whole workflow (3 per default)

### 2.2.2 single job

Here is how to submit a simple job:

```
dbatch --job-name=job_1 my_job.sh
```

```
[MSG  ] submitting job job_1 (for 1) --> Job # job_1-0-1 <-depends-on None
[INFO ] launch-0!0:submitting job job_1 [1] --> Job # job_1-0-1 <-depends-on None
Submitted batch job job_1-0-1
[1] --> Job # job_1-0-1 <-depends-on None
```

Notice how the command syntax is similar to **sbatch** command.

- In lines starting with `[MSG]`, `[INFO]`, or `[DEBUG]`, *Decimate* gives us additional information about what is going on.
- All the traces `[INFO]`, or `[DEBUG]` also appears in the corresponding job output file as well as in *Decimate* central log file dumped in *<current_directory>/.decimate/LOGS/decimate.log* `[MSG]` traces only appears at the console or in the output file of the job.
- for *Decimate*, every job is considered as a job array. In this simple case, it considers an array of job made of a single element `1-1`. In the traces, the array indice shows in "(for **1**)", "submitting job job_1 [**1**]", or "job job_1-0-**1**". (if needed check SLURM job array documentation for more information).
- Every job submitted via *Decimate* is part of a fault-tolerant environment. At the end of its execution, its correctness is systematically checked thanks to a user defined function or by default thanks the return code of the job given by SLURM. If the job is not considered as correct, (and if the return code of the user-defined function is not *ABORT*), the job is automatically resubmitted for a first and a second attempt if needed. In the traces, the attempt number shows as the second figure in the job denomination: "job job_1-**0**-1".

### 2.2.3 dependent job

Here is how to submit a job dependending on a previous job:

```
dbatch --dependency=job_1  --job-name=job_2 my_job.sh
[INFO ] launch-0!0:Workflow has already run in this directory, trying to continue it
[MSG  ] submitting job job_2 (for 1) --> Job # job_2-0-1 <-depends-on 218459
[INFO ] launch-0!0:submitting job job_2 [1] --> Job # job_2-0-1 <-depends-on 218459
Submitted batch job job_2-0-1
[1] --> Job # job_2-0-1 <-depends-on 218459
```

It again matches **sbatch** original syntax with the subtility that via *Decimate* dependency can be expressed with respect to a previous job name and not only to a previous job id as **SLURM** only allows it.

- It makes it more convenient to write automated script.
- At this submission time, *Decimate* checks if a previous submitted job has actually been submitted with this particular name. If not, an error will be issued and the submission is canceled.
- Of course, dependency on a previous job id is also supported.

### 2.2.4 other kind of jobs

A comprehensive list of job examples can be found in Examples of Workflows.

## 2.3 checking the status

The current workflow status can be checked with **dstat**:

```
dstat
```

When no job has been submitted from the current directory. **dstat** shows:

```
[MSG  ] No workflow has been submitted yet
```

When jobs submitted submitted the current directory are currently running . **dstat** shows:

```
[MSG  ] step job_1-0:1-1                SUCCESS   SUCCESS:  100%   FAILURE:   0% ->␣
↪[]
[MSG  ] step job_2-0:1-1                RUNNING   SUCCESS:    0%   FAILURE:   0% ->␣
↪[]
```

And when a workflow is completed:

```
dstat
[MSG  ] CHECKING step : job_2-0 task 1
[MSG  ] step job_1-0:1-1                SUCCESS   SUCCESS:  100%   FAILURE:   0% ->␣
↪[]
[MSG  ] step job_2-0:1-1                SUCCESS   SUCCESS:  100%   FAILURE:   0% ->␣
↪[]
```

## 2.4 Displaying the log file

The current *Decimate* log file can be checked with **dlog**:

```
dlog
```

## 2.5 Cancelling the whole workflow

The current workflow can be completly killed with the command **dkill**:

```
dkill
```

If no job of the workflow is either running, queueing or waiting to be queued, **dkill** prints:

```
[INFO ] No jobs are currently running or waiting... Nothing to kill then!
```

If any job is still waiting or running, *dkill* asks a confirmation to the user and cancels all jobs from the current workflow.

CHAPTER 3

# Examples of Workflows

## 3.1 Test job

Let *my_job.sh* be the following example job:

```
#!/bin/bash
#SBATCH -n 1
#SBATCH -t 0:05:00


echo job running on...
hostname
sleep 10

echo job DONE
```

If not done yet, we first load the *Decimate* module:

```
module load decimate
```

## 3.2 Nominal 2 job workflow

Then submission of jobs follows the same syntax than with the **sbatch** command:

```
dbatch --job-name=job_1 my_job.sh
```

```
[MSG  ] submitting job job_1 (for 1) --> Job # job_1-0-1 <-depends-on None
[INFO ] launch-0!0:submitting job job_1 [1] --> Job # job_1-0-1 <-depends-on None
Submitted batch job job_1-0-1
[1] --> Job # job_1-0-1 <-depends-on None
```

```
dbatch --dependency=job_1  --job-name=job_2 my_job.sh
[INFO ] launch-0!0:Workflow has already run in this directory, trying to continue it
[MSG  ] submitting job job_2 (for 1) --> Job # job_2-0-1 <-depends-on 218459
[INFO ] launch-0!0:submitting job job_2 [1] --> Job # job_2-0-1 <-depends-on 218459
Submitted batch job job_2-0-1
[1] --> Job # job_2-0-1 <-depends-on 218459
```

```
dstat
```

```
[MSG  ] step job_1-0:1-1                  SUCCESS   SUCCESS:  100%   FAILURE:   0% ->␣
↪[]
[MSG  ] step job_2-0:1-1                  RUNNING   SUCCESS:    0%   FAILURE:   0% ->␣
↪[]
```

```
dstat
[MSG  ] CHECKING step : job_2-0 task 1
[INFO ] launch-0!0:no active job in the queue, changing all WAITING in ABORTED???
[MSG  ] step job_1-0:1-1                  SUCCESS   SUCCESS:  100%   FAILURE:   0% ->␣
↪[]
[MSG  ] step job_2-0:1-1                  SUCCESS   SUCCESS:  100%   FAILURE:   0% ->␣
↪[]
```

## 3.3 parametric job workflow

Then submission of parametric jobs follows the same syntax than with the **sbatch** command adding a reference to a text file describing the set of parameters to be tested:

```
dbatch --job-name=job_1 -P parameters.txt my_job.sh
```

How to build the file *parameters.txt* is described at Parameters combination.

CHAPTER 4

# Parameters combination

Then submission of parametric jobs requires to gather in a *parameter* file all the combinations of parameters that one wants to run a job against. This list of combination can be described as an explicit array of values of programatically via a Python or shell script or using simple directives.

While the execution of parametric workflows is described here, here are detailed four ways of defining parameters. .

## 4.1 array of values

The simplest way to describe the set of parameter combinations that needs to be tested consists in listing them extensively as an array of values. The first row of this array is the name of each parameters and each row is one possible combination.

Here is a parameters file listing all possible combinations for 3 parameters (i,j,k), each of them taking the value 1 or 2.

```
# array-like description of parameter combinations

i   j   k

1   1   1
1   1   2
1   2   1
1   2   2
2   1   1
2   1   2
2   2   1
2   2   2
```

Notice that:

- spaces, void lines are ignored.
- every thing following a # is considered as a comment and ignored

## 4.2 Combined parameter sweep

In case of combinations that sweeps all possible set of values based on the domain definition of each variable, a more compact declarative syntax is also available. The same set of parameters can be generated with the following file:

```
# combine-like description of parameter combinations

#DECIM COMBINE i = [1,2]
#DECIM COMBINE j = [1,2]
#DECIM COMBINE k = [1,2]
```

Every line starting with *#DECIM* is parsed as a special command.

## 4.3 Parameters depending on simple formulas

Some parameters can also be computed from others using simple arithmetic formulas. Here is a way to declare them:

```
# combine-like description of parameter combinations

#DECIM COMBINE i = [1,2]
#DECIM COMBINE j = [1,2]
#DECIM COMBINE k = [1,2]

#DECIM p = i*j*k
```

which is a short way to describe the same 8 combinations as expressed in the following array-like parameter file:

```
# array-like description of parameter combinations

i  j  k  p

1  1  1  1
1  1  2  2
1  2  1  2
1  2  2  4
2  1  1  2
2  1  2  4
2  2  1  4
2  2  2  8
```

an additional parameter can also be described by a list of values:

```
# combine-like description of parameter combinations

#DECIM COMBINE i = [1,2]
#DECIM COMBINE j = [1,2]
#DECIM COMBINE k = [1,2]

#DECIM p = i*j*k

#DECIM t = [1,2,4,8,16,32,64,128,256]
```

which is a short way to describe the same 8 combinations as expressed in the following array-like parameter file:

```
# array-like description of parameter combinations

i  j  k  p    t

1  1  1  1    1
1  1  2  2    2
1  2  1  2    4
1  2  2  4    8
2  1  1  2   16
2  1  2  4   32
2  2  1  4   64
2  2  2  8  128
```

For each parameter added via a list of values, the conformance with the existing number of already possible combinations is checked. For example, the following parameter file...

```
# combine-like description of parameter combinations

#DECIM COMBINE i = [1,2]
#DECIM COMBINE j = [1,2]
#DECIM COMBINE k = [1,2]

#DECIM p = i*j*k

#DECIM t = [1,2,4,8,16,32,64,128,256]
```

...produces the error:

```
[ERROR] parameters number mistmatch for expression
[ERROR]       t = [1,2,4,8,16,32,64,128,256]
[ERROR]       --> expected 8 and got 9 parameters...
```

## 4.4 More complex Python expressions

For a high number of parameters, a portion of code written in Python can also be embedded after a *#DECIM PYTHON* directive till the end of the file.

```
# pythonic parameter example file

#DECIM COMBINE nodes = [2,4,8]
#DECIM COMBINE ntasks_per_node = [16,32]

#DECIM k = range(1,7)

#DECIM PYTHON

import math

ntasks = nodes*ntasks_per_node
nthreads = ntasks * 2

NPROC = 2; #Number of processors

t = int(2**(k))
T = 15
```

which is a short way to describe the same 8 combinations as expressed in the following array-like parameter file:

```
# array-like description of parameter combinations

nodes  ntasks_per_node  k  ntasks  nthreads   t  NPROC    T
   2               32  1      64       128   2      2   15
   2               64  2     128       256   4      2   15
   4               32  3     128       256   8      2   15
   4               64  4     256       512  16      2   15
   8               32  5     256       512  32      2   15
   8               64  6     512      1024  64      2   15
```

A python section is always evaluated at the end. Each new variables set at the end of the evaluation is added as a new parameter computed against each of the already built combinations. The conformance to the number of combinations already set is also checked if the variable is a set of values.

CHAPTER 5

# Shell API

## 5.1 dbatch

Usage: dbatch [OPTIONS. . . ] job_script [args. . . ]

Help:

**-h, --help** show all possible options for **dbatch**

**-H, --decimate-help** show hidden option to manage *Decimate* engine

Workflow management:

**--check=SCRIPT_FILE** python or shell to check if results are ok

**--max-retry=MAX_RETRY** number of time a step can fail and be restarted automatically before failing the whole workflow (3 per default)

Burst Buffer:

**-bbz, --use-burst-buffer-size** use a non persistent burst buffer space

**-xz, --burst-buffer-size=BURST_BUFFER_SIZE** set Burst Buffer space size

**-bbs, --use-burst-buffer-space** use a persistent burst buffer space

**-xs, --burst-buffer-space=BURST_BUFFER_SPACE_name** sets Burst Buffer name

environment variables:

DPARAM options forwarded to Decimate

script directives

#DECIM SHOW_PARAMETERS #DECIM PROCESS_TEMPLATE_FILES

CHAPTER 6

# Installation

## 6.1 How to submit