Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Fred Stober, H. Stadie, C. Garbers,
N. Kovalchuk, P. Schleper, M. Fischer

Karlsruher Institut für Technologie

grid-control

The Swiss Army knife of job submission tools

What is grid-control?

- grid-control is an open source **job submission tool**
- It supports **all aspects** of a common HEP user analysis:
  - working with official experiment software with multiple generation or processing steps
  - doing ntuple analysis with ROOT macros
  - running all different kinds of custom software
- grid-control itself can run in virtually **any environment** and is trivial to deploy, making it a useful tool in the context of long term analysis efforts
- The software allows job submission to a wide range of **local or remote** workflow management software. For small tasks or tests, it can run jobs on the host itself.
- It is built around a **powerful plugin and configuration system**, that allows to provide additional functionality and customize the tool for the particular site / user / task / experiment
- With grid-control, it is very easy to define tasks to process **complex parameter spaces**, that can encompass automatically **partitioned datasets**.

Configuration and Execution

grid-control can be used in conjunction with **ini-style** or **python config files** that are given to the command line program or by **calling the API** from a user script. The program is commonly run from within the GNU **screen** tool.

Command line usage:

Keep running and resubmitting until the workflow is finished:

```
gridcontrol -c <config file>
```

Keep running and resubmit failing jobs up to 3 times:

```
gridcontrol -c -m 3 <config file>
```

Cancel all jobs:

```
gridcontrol -d ALL <config file>
```

Workflow execution:

While grid-control is running, it will check the status of running jobs, retrieve and analyse finished jobs and submit jobs that are ready. When the continuous mode (-c) is active, grid-control will repeat this until the task is finished.

Basic example

This is a simple example that shows how to configure grid-control to run a script on the local machine and write "Hello World" to an output file.

HelloWorld.conf

```
[global]
task = UserTask # run user scripts
backend = Host  # jobs will run locally
[jobs]
jobs     = 1    # number of jobs to run
walltime = 0:01 # anticipated run time
[task]
executable = test.sh # path to script
arguments  = World   # script arguments
```

test.sh

```
#!/bin/sh
echo Hello $@
```

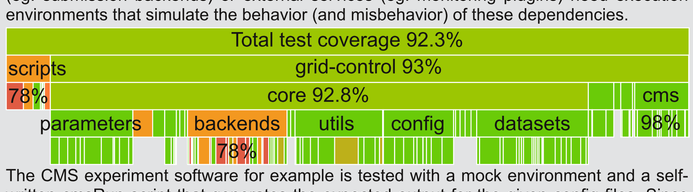
```
> gridcontrol -c HelloWorld.conf
Current task ID: GCdbf7bebe81b5
<...>
Running in continuous mode. Press ^C to exit.
2016-10-01 14:00:00 - Job 0 state changed from INIT to SUBMITTED
2016-10-01 14:00:01 - Job 0 state changed from SUBMITTED to RUNNING
2016-10-01 14:00:02 - Job 0 state changed from RUNNING to DONE
2016-10-01 14:00:03 - Job 0 state changed from DONE to SUCCESS
2016-10-01 14:00:03 - Task successfully completed. Quitting...
> zcat work.HelloWorld/output/job_0/job.stdout.gz
Hello World
```

Development

Development of grid-control started February 9th 2007. A week later on February 16th, the first version with support for submitting CMSSW grid jobs was released. grid-control has been constantly evolving ever since and continues to be open to contributions on github.
Despite the large set of functionality, and neglecting 3rd party libraries, grid-control still only consists of less than **20k lines of code**.

Unit Tests

The source code is automatically tested with **Travis CI** using the default linux docker container with CPython 2.6-3.5 and pypy/pypy3. Tests with older versions (Python 2.3-2.5) are performed manually on a regular basis. The code coverage is reviewed with the codecov.io service, while code convention and quality checks are done with landscape.io. All services are integrated into the development workflow on github.
Achieving a high code coverage is difficult, since code that depends on external software (eg. submission backends) or external services (eg. monitoring plugins) need execution environments that simulate the behavior (and misbehavior) of these dependencies.



The CMS experiment software for example is tested with a mock environment and a self-written cmsRun script that generates the expected output for the given config files. Since the CMS data services require authentication with a grid proxy, the CMS data provider plugins are tested with a webservice query plugin that responds with pre-recorded answers. The REST query plugins are tested against the public httpbin.org service.

Code structure

The source code is organized into packages, with `grid_control` being the core package. Other packages (like `grid_control_cms`) contain plugins that replace or extend the functionality provided by the core. The packages are completely encapsulated - and as long as the user didn't specify a plugin from the package, it can be safely deleted from the tree.

Documentation

In addition to a large set of examples that cover most use cases and a small tutorial, grid-control comes with a comprehensive list of all available configuration options (currently more than 530) together with their default value, type, and scope. This list is created by a custom script directly from the python AST and ensures that no option in the source code is left undocumented.

Parameterized jobs

A particularly useful feature for HEP applications is the sophisticated job parameter system built into grid-control. It provides a convenient way to define the parameter space that a task is based on. This parameter space can be build up from any number of variables and data sources, which can have complex dependencies on each other.
It is also possible to specify the job requirements (like wall time / memory) for each parameter space point separately.

```
[global] include = HelloWorld.conf
[parameters]
parameters = (MUR,MUF) VAR[MUR] + {pspace1}
(MUR,MUF) = (1,1) (2,1) (1,2)
VAR = def
  2 => x y
[pspace1]
parameters = MUR MUF
MUR = 1 2
MUF = 0.5
```

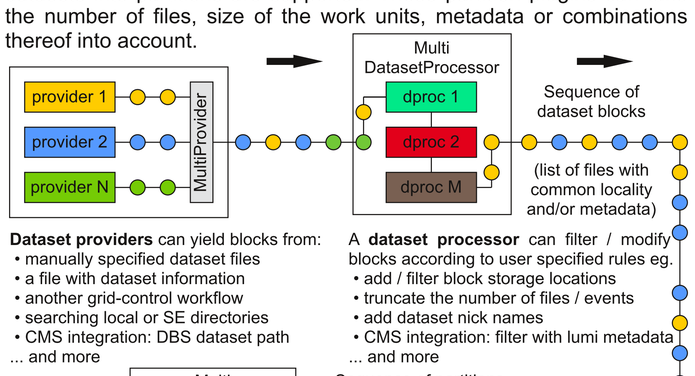
```
MUR = 0.5 1
MUF = 0.5
```

Job	MUR	MUF	VAR
0	1	1	def
1	2	1	x
2	2	1	y
3	1	2	def
4	1	0.5	
5	2	0.5	
6	0.5	0.5	

The parameter system is able to handle changes to the data source as well as to other parameters. It can transparently adapt the job submission to the new parameter space at runtime.
Using the parameters on the bottom right, after previously starting grid-control with the bottom left parameters, will disable (and cancel if needed) job 5 and submit a new job 6.

Dataset processing pipeline

grid-control provides several plugins to retrieve datasets from different sources. These datasets may contain a list of URLs, an optional number of work units (eg. events or file size), arbitary metadata (eg. lumi section information or generator parameters) and locality information.
All datasets are processed through a configurable pipeline of dataset filters, partition plugins and partition filters. Several methods to split datasets into partitions are supported. These partition plugins can take the number of files, size of the work units, metadata or combinations thereof into account.

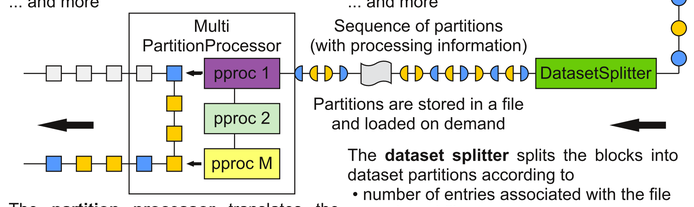


Dataset providers can yield blocks from:
- manually specified dataset files
- a file with dataset information
- another grid-control workflow
- searching local or SE directories
- CMS integration: DBS dataset path

... and more

A **dataset processor** can filter / modify blocks according to user specified rules eg.
- add / filter block storage locations
- truncate the number of files / events
- add dataset nick names
- CMS integration: filter with lumi metadata

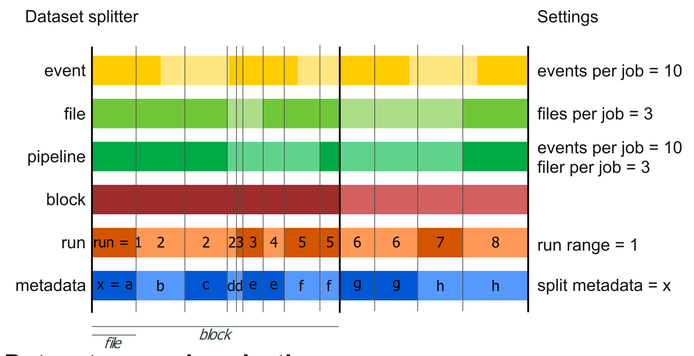
... and more



The **partition processor** translates the partition information into the variables and requirements of a parameter space point.
It can also perform operations that are similar to the work done by the dataset processor (eg. change LFNs into PFNs)

The **dataset splitter** splits the blocks into dataset partitions according to
- number of entries associated with the file
- number of files
- classification of file metadata information
- CMS integration: run / lumi metadata

... and more / *combinations of the above*



Dataset resynchronization

Changes to the dataset on file level (additions or removals) as well as on the work unit level (expanding or shrinking files), are propagated through the processing pipeline and **transparently trigger adjustments** to the processed parameter space.
For HEP workflows this allows to run a **"live"** analysis on an ever expanding dataset with a single processing task that regularly queries some datasource and spawns new jobs.

Example

The information from the dataset file "example.dbs" will yield 3 jobs (with file1+file2, file3, fileA). The environment variable GEN_PROCESS will be set to "Zmumu" for the first 2 jobs. Every 1 hour, the file "example.dbs" will be re-read and the specified directory will be scanned and jobs created / disabled if new files are found / files are missing.

```
[global] include = HelloWorld.conf
[UserTask]
dataset =
  example.dbs
  myname : scan : srm:/storage/path
files per job = 2
dataset refresh = 1:00
partition processor += metadata
partition metadata = GEN_PROCESS
```

example.dbs

```
[/PRIVATE/MC1#42]
metadata = ["GEN_PROCESS"]
metadata common = ["Zmumu"]
/path/to/file1 = 1000
/path/to/file2 = 2000
/path/to/file3 = 1000
[/PRIVATE/MC2]
/path/to/fileA = 3000
```

Runtime support / monitoring

grid-control provides the user with several tools to make it easier and safer to run jobs. It can substitute variables in configuration files where needed, monitor the disk usage, automatically select an appropriate scratch directory and perform the stage-in and stage-out of files. For such storage operations, it is possible to specify multiple stage-out destinations to spread the load of the file transfers (however close SEs are always preferred).

Requirements / Installation

Requirements on the submission host:

- CPython 2.3 – 3.5 / pypy 2.4 – 5.3 with python standard libraries
- (optional) requests – for faster interactions with webservies
- (optional) cherrypy – for web based user interface
- (highly recommended) a supported submission backend (ARC, Condor, CreamCE, glite/-wms, PBS, LSF, SLURM, JMS, GridEngine)

Requirements on the worker nodes:

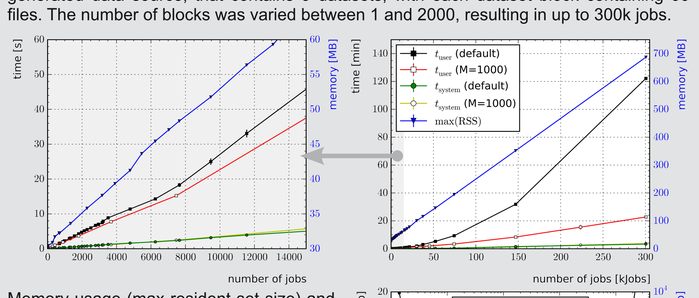
- bash, GNU coreutils, awk, sed
- (optional) gfal / other supported transfer tools – for storage access
- (optional) cvmfs – to allow running the LHC experiment software

Installation:

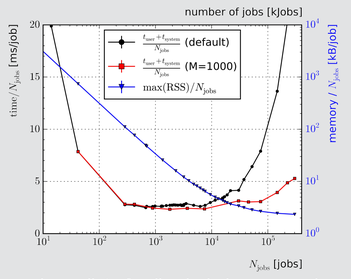
- `pip install (--user) grid-control`

Performance

These figures show the scaling behavior of grid-control for a workflow using a **single core** on an i5 2.4GHz machine without SSD. Only the time used by grid-control itself is measured. The workflow is configured to run with 3 different parameters over a randomly generated data source, that contains 9 datasets, with each dataset block containing 99 files. The number of blocks was varied between 1 and 2000, resulting in up to 300k jobs.

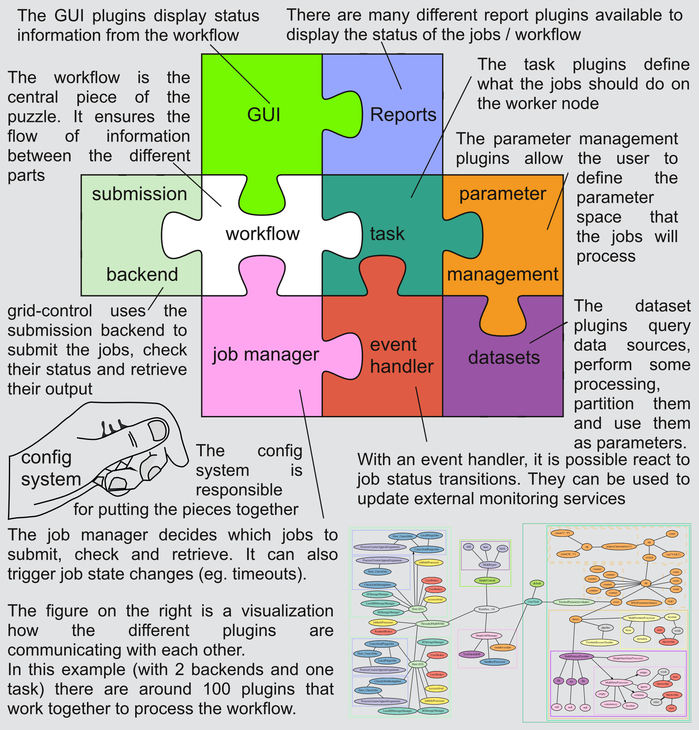


Memory usage (max resident set size) and system time scales linearly between 1 and 300k jobs. In addition to 30 MB per task, the used memory scales with **2.3 kb/job** since the default job management plugin keeps all job information also in memory.
With an initial **0.25 s startup time**, the time spend by grid-control itself on a single job over it's whole lifetime is around **2.3 ms/job** for most task sizes.
By default, the list of jobs with size N, that is selected for submission, status query or retrieval is sorted before a chunk of size M is processed in one pass. In order to give users immediate feedback during processing, the default chunk size is M = 100 jobs, which leads to a linear relationship between the processing time and N for N < 10k jobs. This chunk size can be configured (shown is a chunk size of M = 1k jobs), which allows to keep the linear behavior for N < 100k jobs.



Design and Architecture

grid-control is using a **highly modular design**, where all functionality is provided by specialized plugins. The core package provides around 300 plugins, most of which belong to one of 10 larger categories.
The CMS experiment software integration (with full support for CMSSW version 1.x-8.x, DAS/DBS3/PhEDEx) is done with ~20 small plugins.
It is very **easy to replace any plugin** with a custom implementation that is adapted to the particular needs of the user or adds new functionality.
The interplay between the different plugin categories is shown below.



The figure on the right is a visualization how the different plugins are communicating with each other.
In this example (with 2 backends and one task) there are around 100 plugins that work together to process the workflow.

These developers contributed code to grid-control:

https://github.com/grid-control/grid-control