CI-HPC Documentation

Jan Hybs

Jan 08, 2019

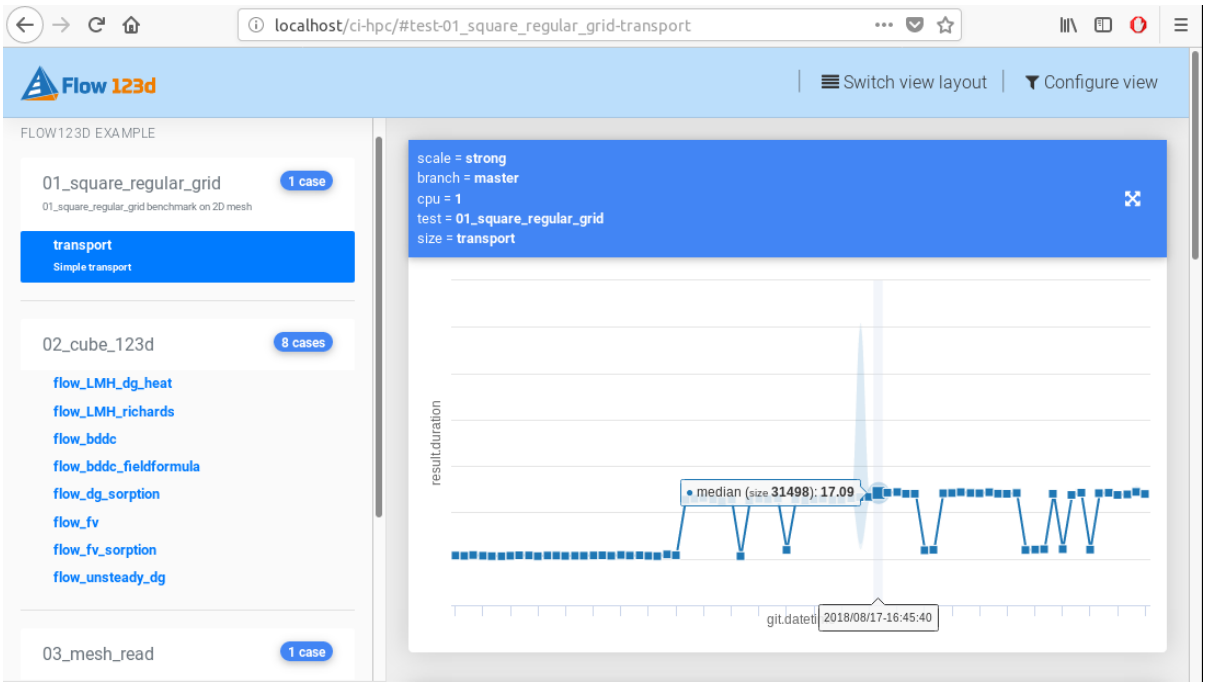
CI-HPC Documentation

A simple framework which monitors a performance and scalability of software packages. The framework presented here combines Continuous Integation & High Performance Computing together with a minimalistic set of Python scripts. The results can be visualised in form of static Jupyter notebook or in an interactive web page.

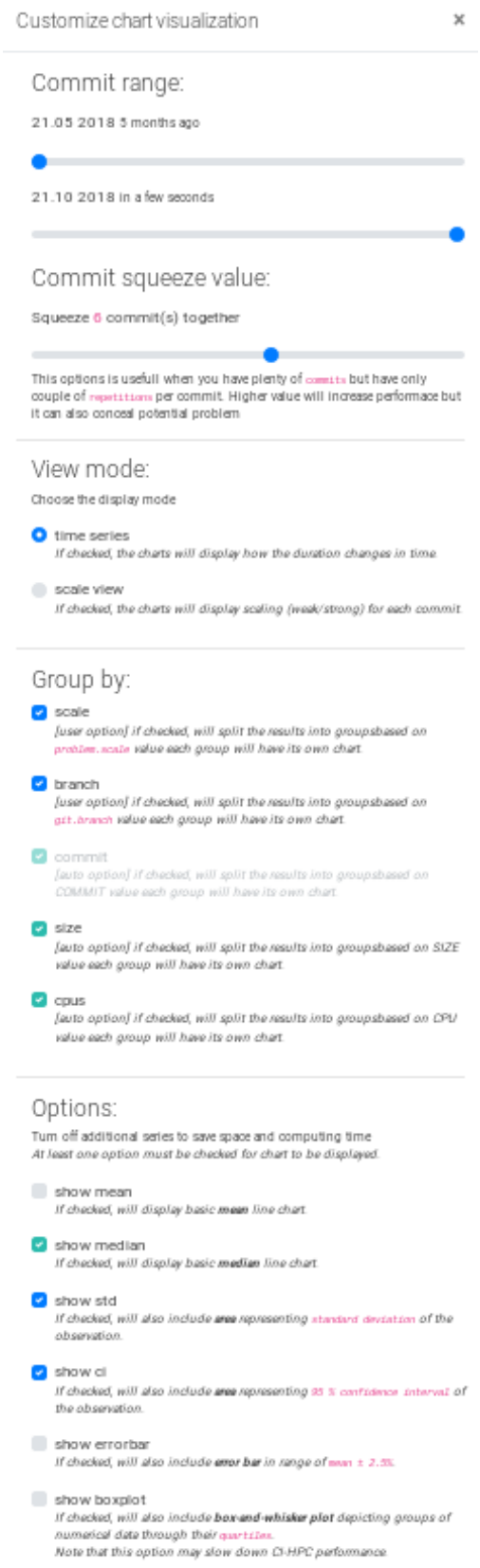
CHAPTER 1

Showcase

1.1 Visualisation page



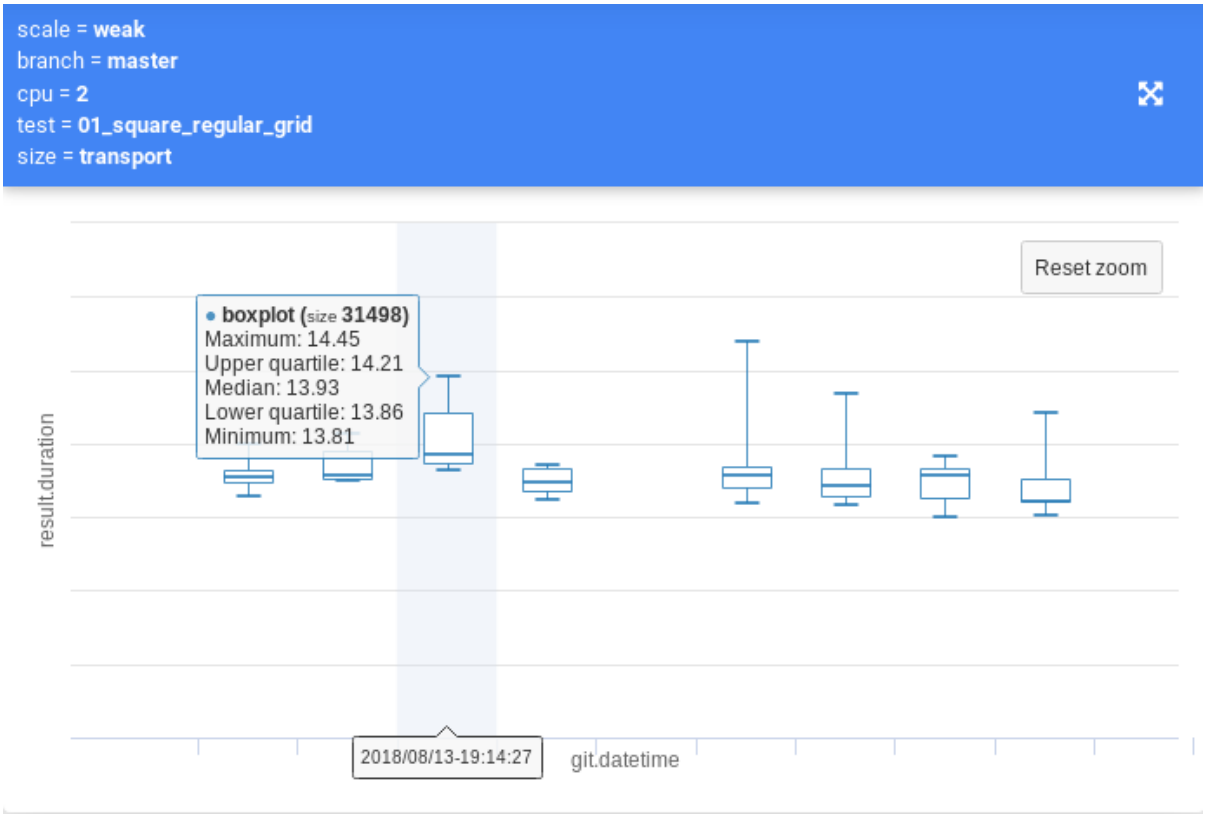
1.2 View configuration



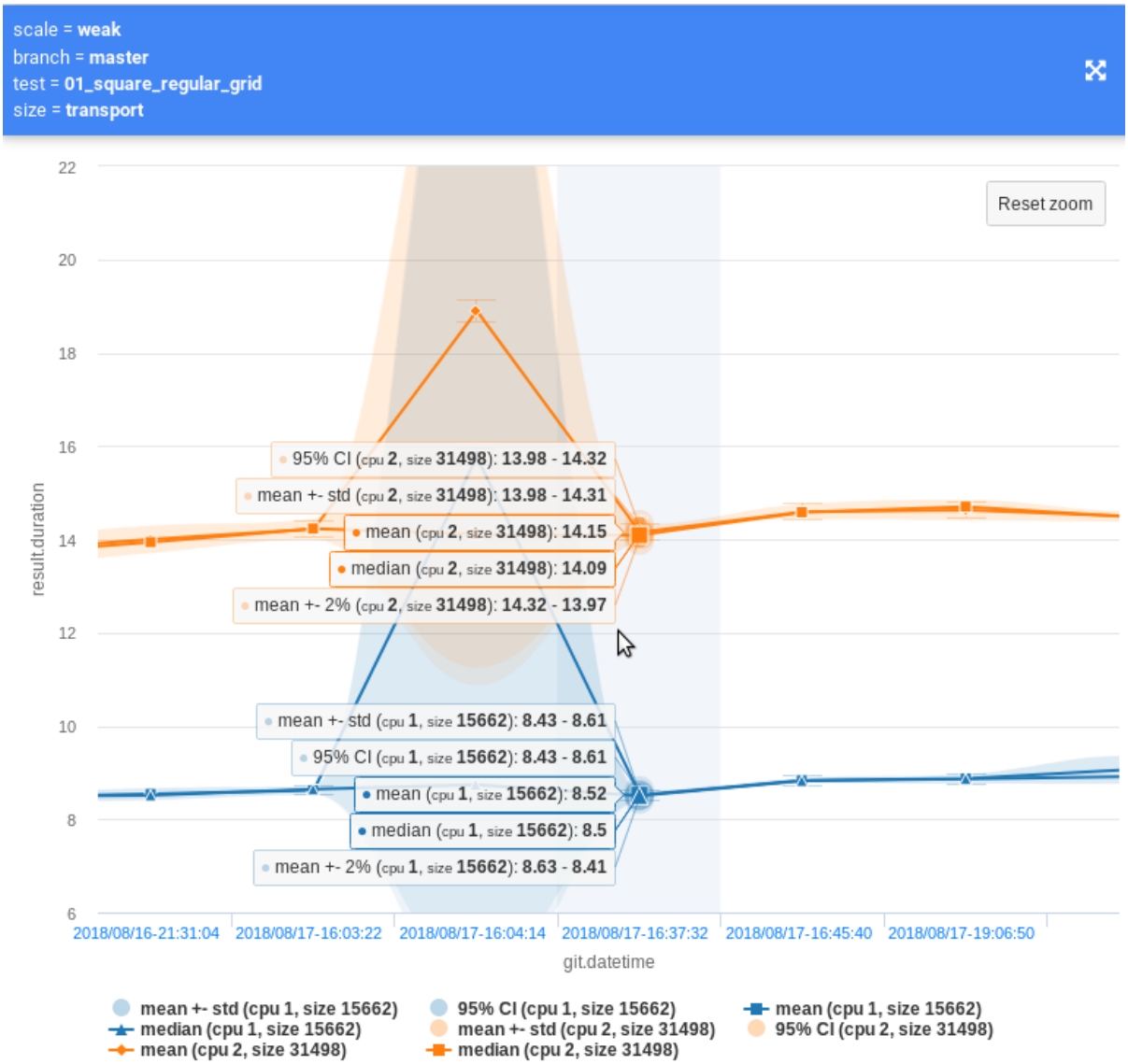
1.3 Zoom detail


scale = strong
branch = master
cpu = 1
test = 01_square_regular_grid
size = transport
Reset zoom
median (size 31498): 16.36
result.duration
2018/08/08-11:59:48
t.datetime

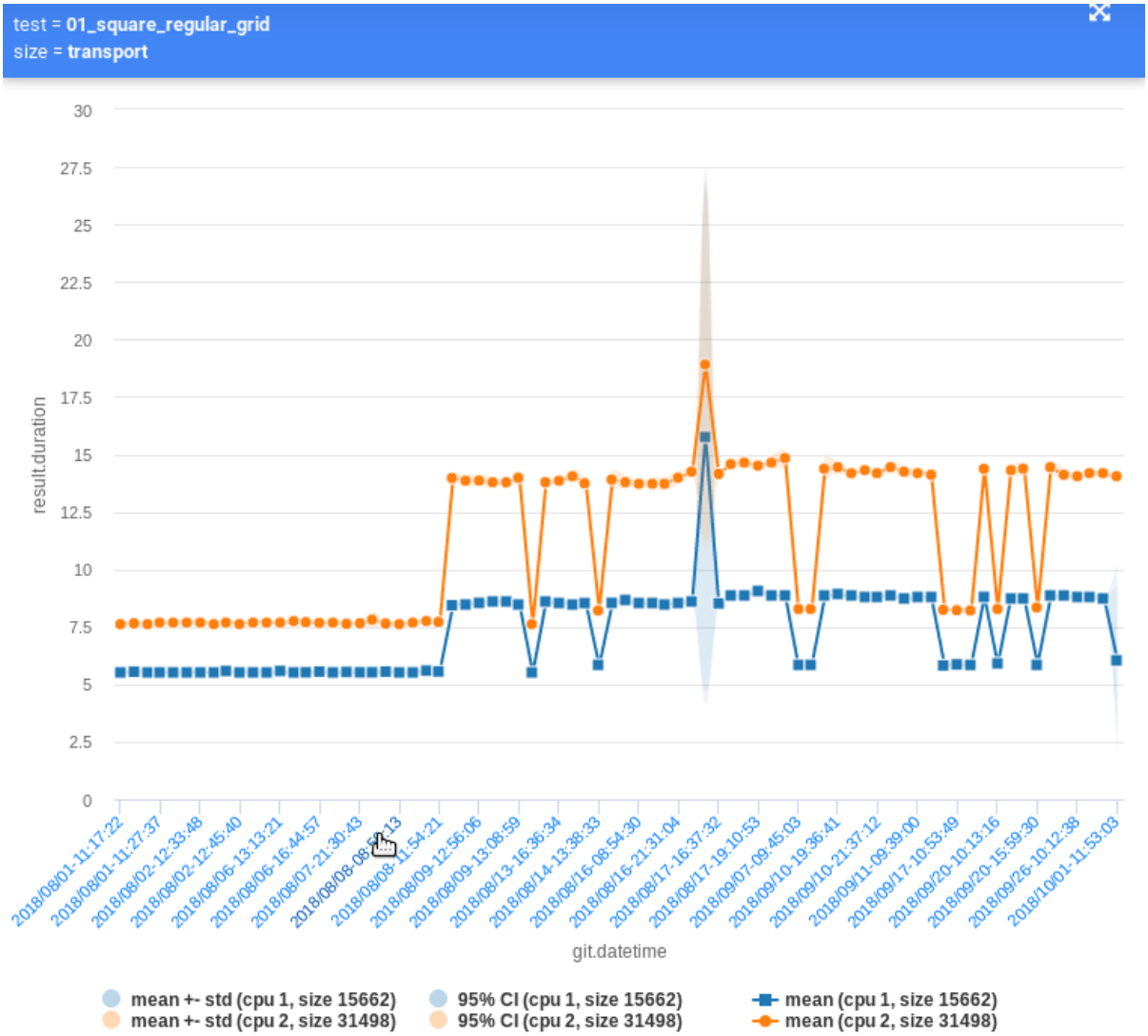
1.4 Boxplot view chart



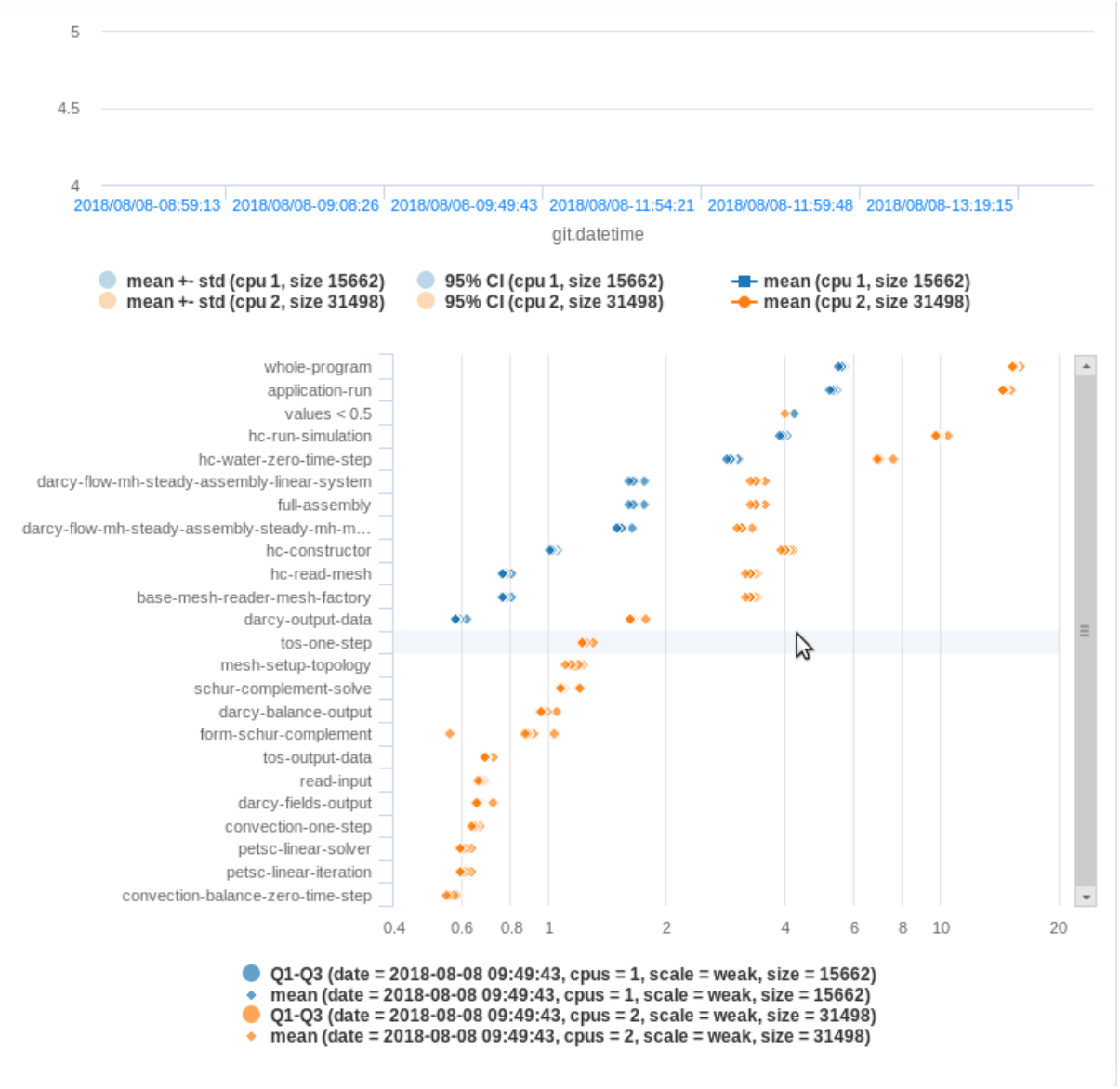
1.5 Detail view

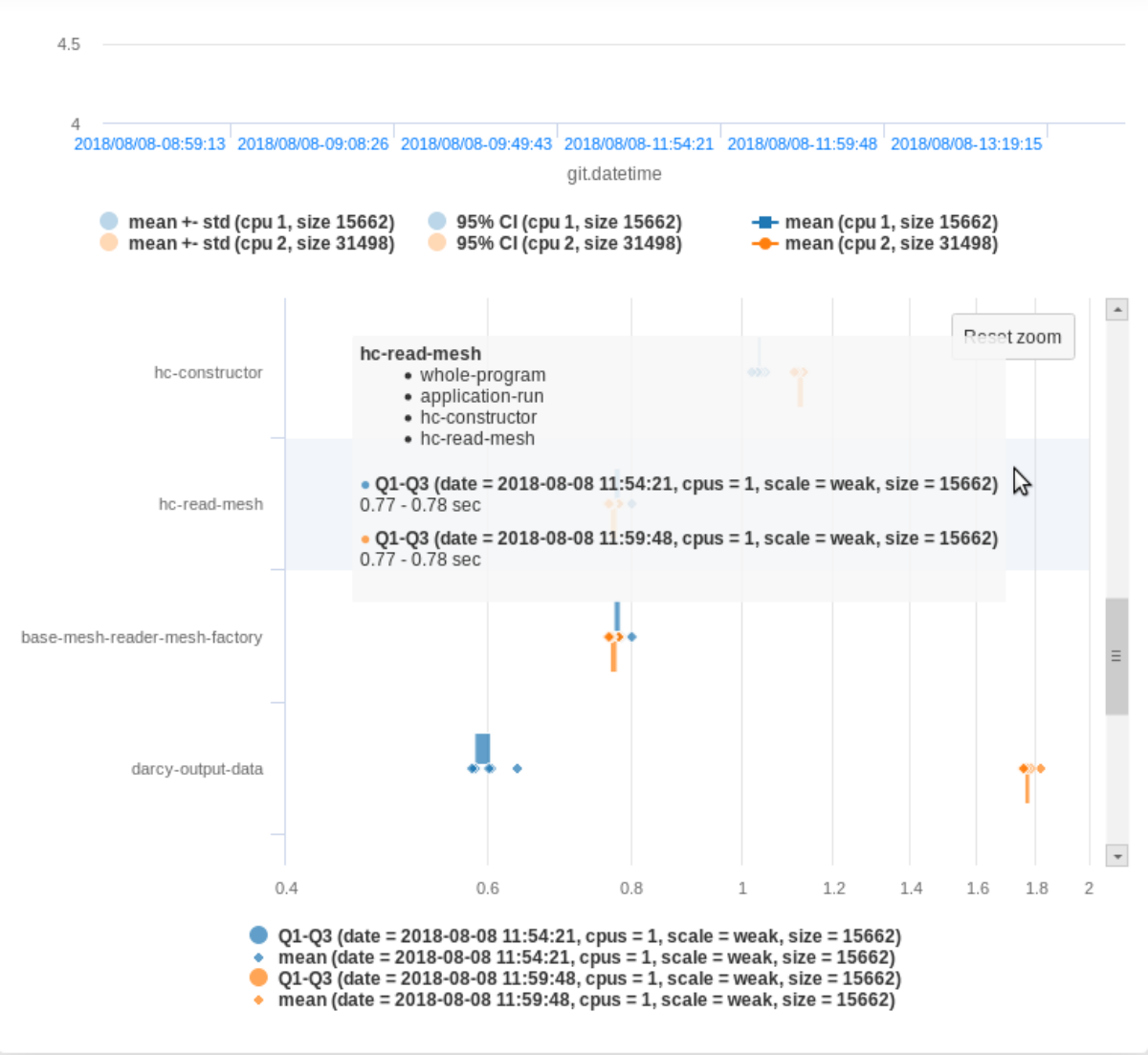


1.6 Link to commit

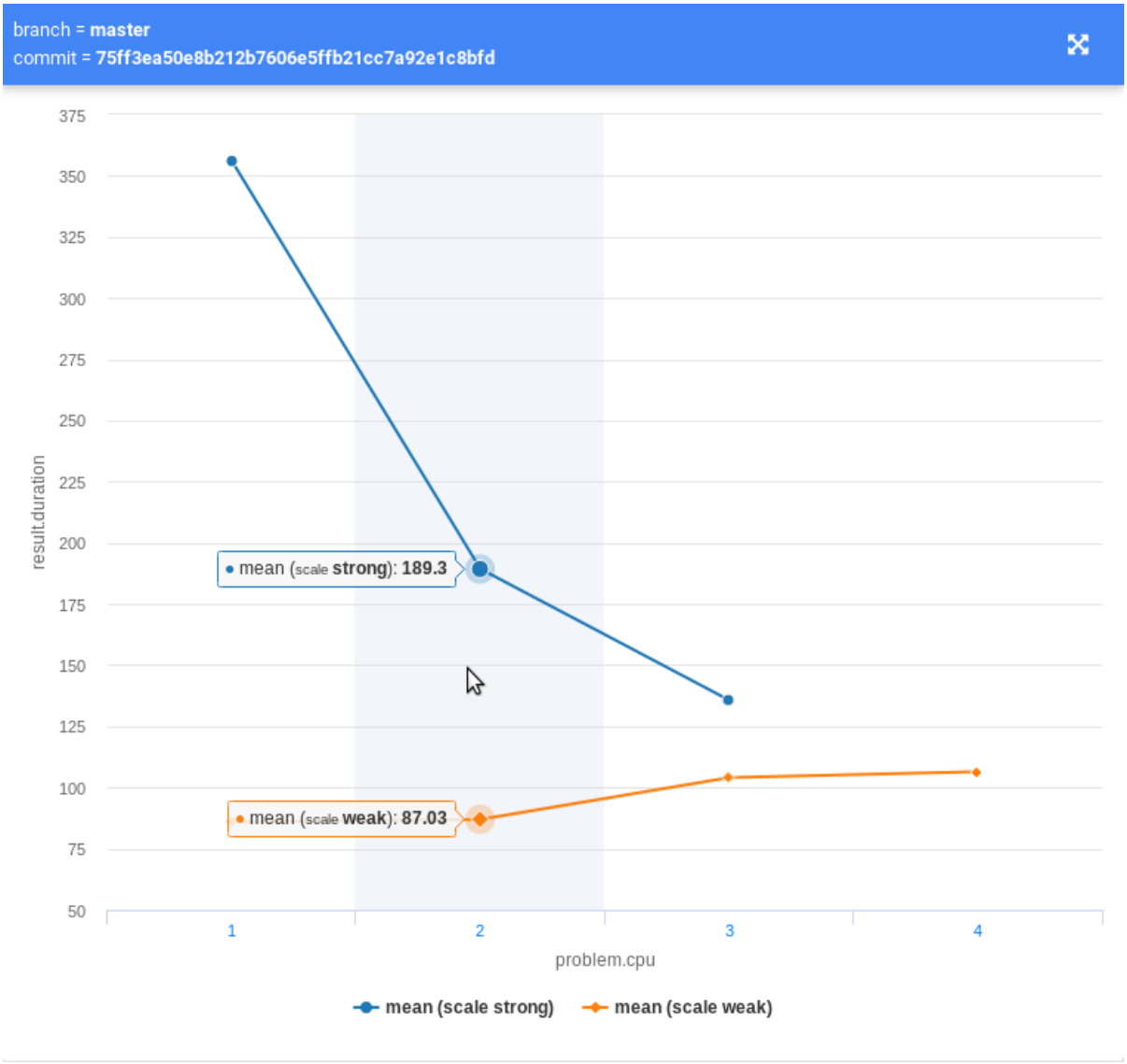


1.7 Frame breakdown

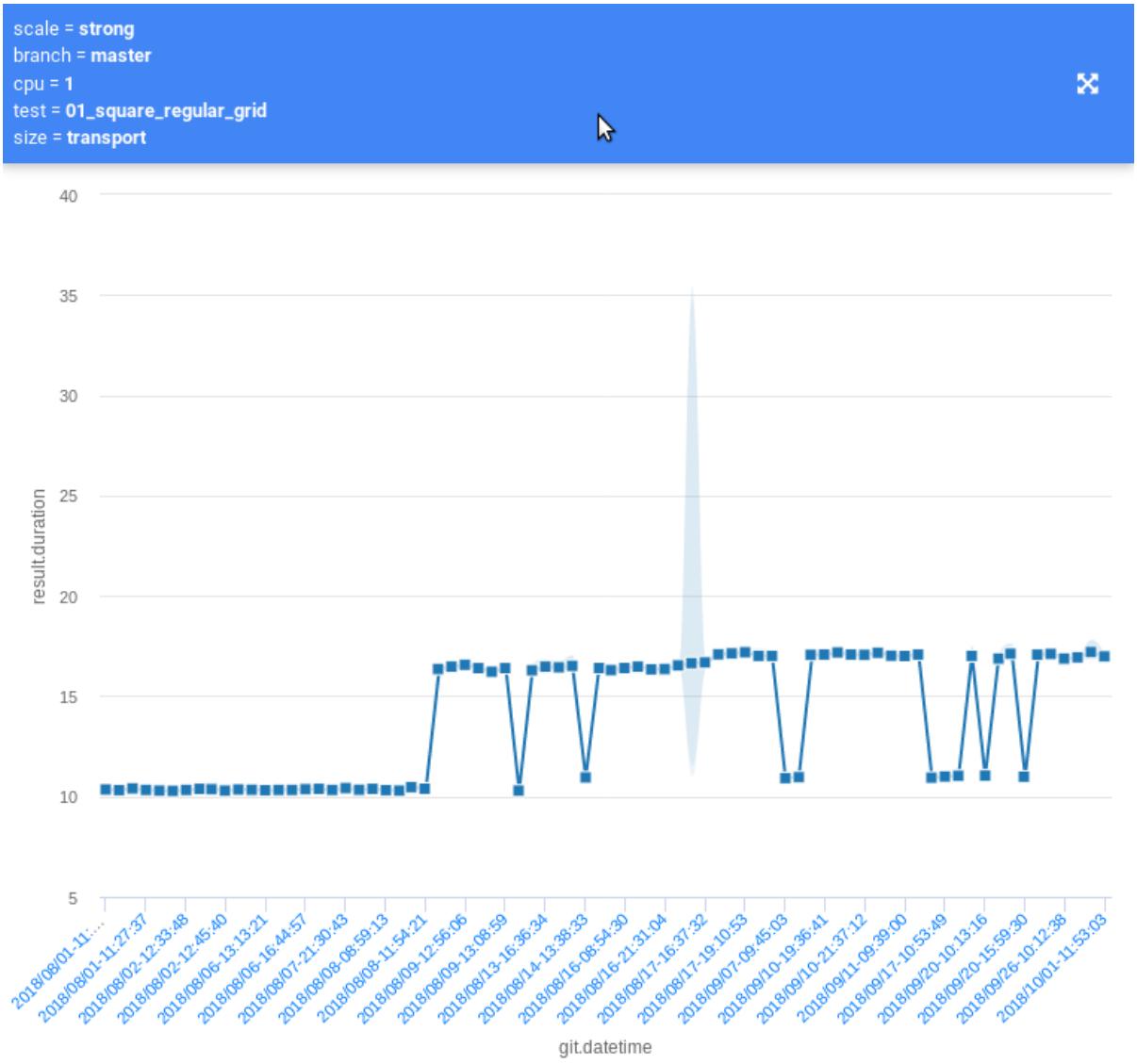


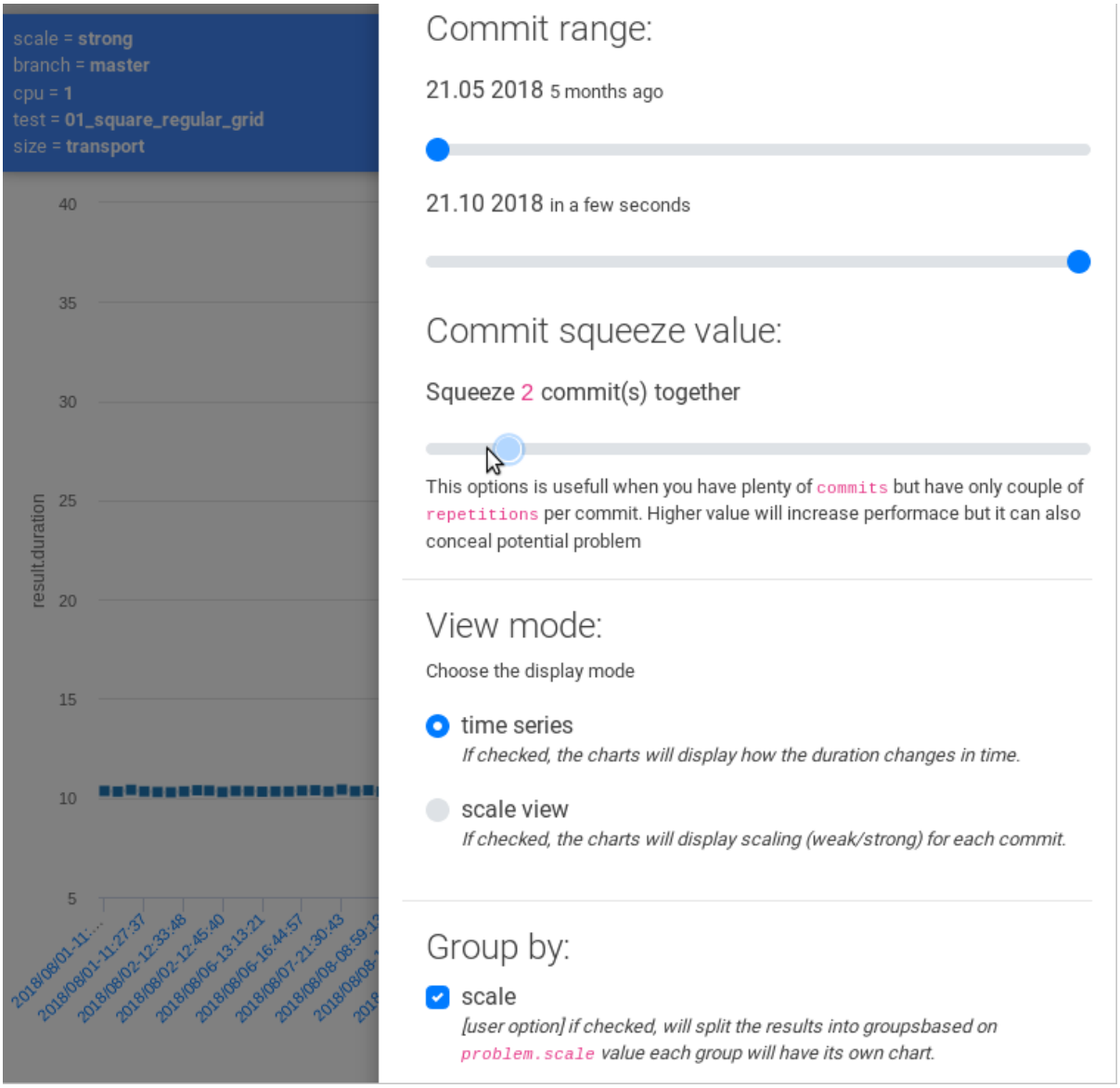

4.5
4
2018/08/08-08:59:13
2018/08/08-09:08:26
2018/08/08-09:49:43
2018/08/08-11:54:21
2018/08/08-11:59:48
2018/08/08-13:19:15
git.datetime
mean +- std (cpu 1, size 15662)
mean +- std (cpu 2, size 31498)
95% CI (cpu 1, size 15662)
95% CI (cpu 2, size 31498)
mean (cpu 1, size 15662)
mean (cpu 2, size 31498)
Reset zoom
hc-constructor
hc-read-mesh
base-mesh-reader-mesh-factory
darcy-output-data
hc-read-mesh
whole-program
application-run
hc-constructor
hc-read-mesh
Q1-Q3 (date = 2018-08-08 11:54:21, cpus = 1, scale = weak, size = 15662)
0.77 - 0.78 sec
Q1-Q3 (date = 2018-08-08 11:59:48, cpus = 1, scale = weak, size = 15662)
0.77 - 0.78 sec
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Q1-Q3 (date = 2018-08-08 11:54:21, cpus = 1, scale = weak, size = 15662)
mean (date = 2018-08-08 11:54:21, cpus = 1, scale = weak, size = 15662)
Q1-Q3 (date = 2018-08-08 11:59:48, cpus = 1, scale = weak, size = 15662)
mean (date = 2018-08-08 11:59:48, cpus = 1, scale = weak, size = 15662)

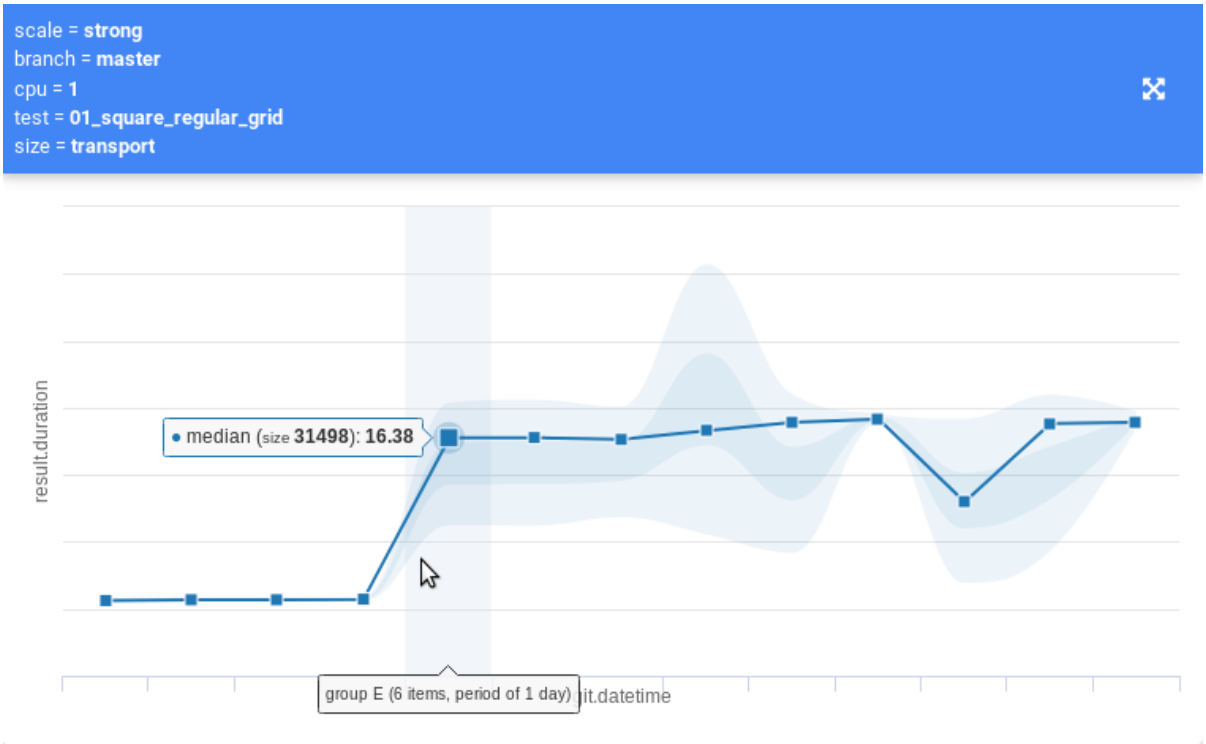
1.8 Scaling mode view



1.9 Commit squeeze




scale = strong
branch = master
cpu = 1
test = 01_square_regular_grid
size = transport
40
35
30
25
20
15
10
5
result.duration
Commit range:
21.05 2018 5 months ago
21.10 2018 in a few seconds
Commit squeeze value:
Squeeze 2 commit(s) together
This options is usefull when you have plenty of commits but have only couple of repetitions per commit. Higher value will increase performace but it can also conceal potential problem
View mode:
Choose the display mode
time series
If checked, the charts will display how the duration changes in time.
scale view
If checked, the charts will display scaling (weak/strong) for each commit.
Group by:
scale
[user option] if checked, will split the results into groupsbased on problem.scale value each group will have its own chart.



Todo: Add image captions and descriptions

CHAPTER 2

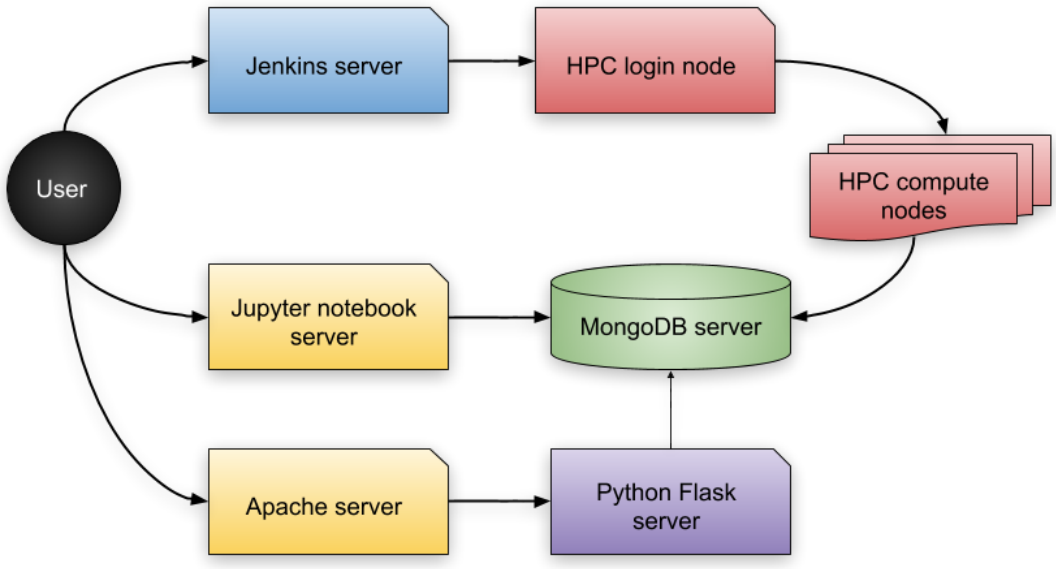
CI-HPC Documentation & Installation

Installation process is not that simple, so sadly **you won't find here something like this**:

```
./configure && make && make install
```

Perhaps in the future version, installation will be easier… but if you know, what you are doing, you can setup your project within 15 minutes.

In order to install CI-HPC framework, please understand its structure first:



From the illustration above, you can see there are several servers.

1. With BLUE color is a Jenkins server. This server is in charge of git repository checking. If Jenkins detects any change in repository, it will contact an HPC login node to starts off the installion and testing of your project.
2. HPC system (in a RED color) consist of 2 parts:
   (a) The *login* node will translates what is Jenkins trying to do and will prepare a PBS job, that will install your project and after that run your benchmark for your project.
   (b) The *compute* nodes, that will take care of the installation and testing and when they are done, they will store the results to a database.
3. The database server (in a GREEN color) has a MongoDB database running and stores and loads benchmark results.
4. You have 2 options when it comes to visualising your results (both options are marked with YELLOW color):
   (a) The first option (slightly easier but not by much) is (*probably soon to be deprecated* Jupyter Notebook server. This solution offers great customization but requires knowledge about Python and some python's scientific packages.
   (b) The seconds option, interactive website, offers more interactivity and better visualisation. Thanks to `highcharts` js framework, you have plenty of options for your charts. You can zoom in thre results, filter the series or simply (by clicking) go to the commit, which you are interested in.
5. Along interactive website, you need to have additional server running (the data need to get to the web page somehow), and this is why there is this last server (in PURPLE color). It has a `python flask` server running, which is serving the data from the database back to the website.

2.1 Prerequisites

Before configuring anything, make sure you have:

1. an access to the HPC node (login preferable via `SSH Key-Based Authentication`).
2. an access to a CI server such as Jenkins or other similar tool. If you have no such server available, `CRON` *may* suffice.
3. an access to the database server, for now only MongoDB is supported. You can get free hosting on MongoDB Atlas for up to 500MB.
4. an access to a jupyter notebook server for visualisation. For education purposes Azure notebooks is possible option.

   or

   access to a web server and access to a flask server. Flask server can be isntalled easily via `pip` packaging tool.

Note: Jenkins server, Database server and visualisation servers can be running on single computer.

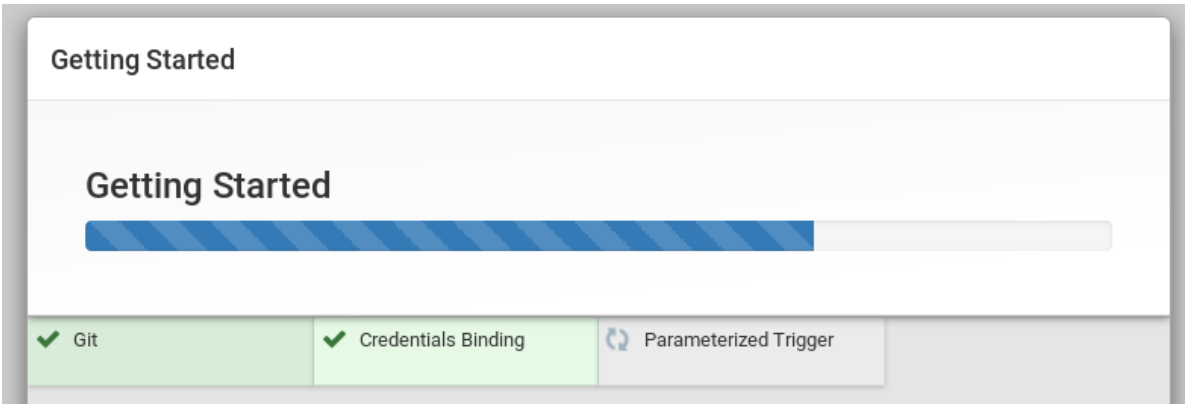
CHAPTER 3

Installing a Jenkins server

1. Install Jenkins server on your server manually or use docker solution like this or this

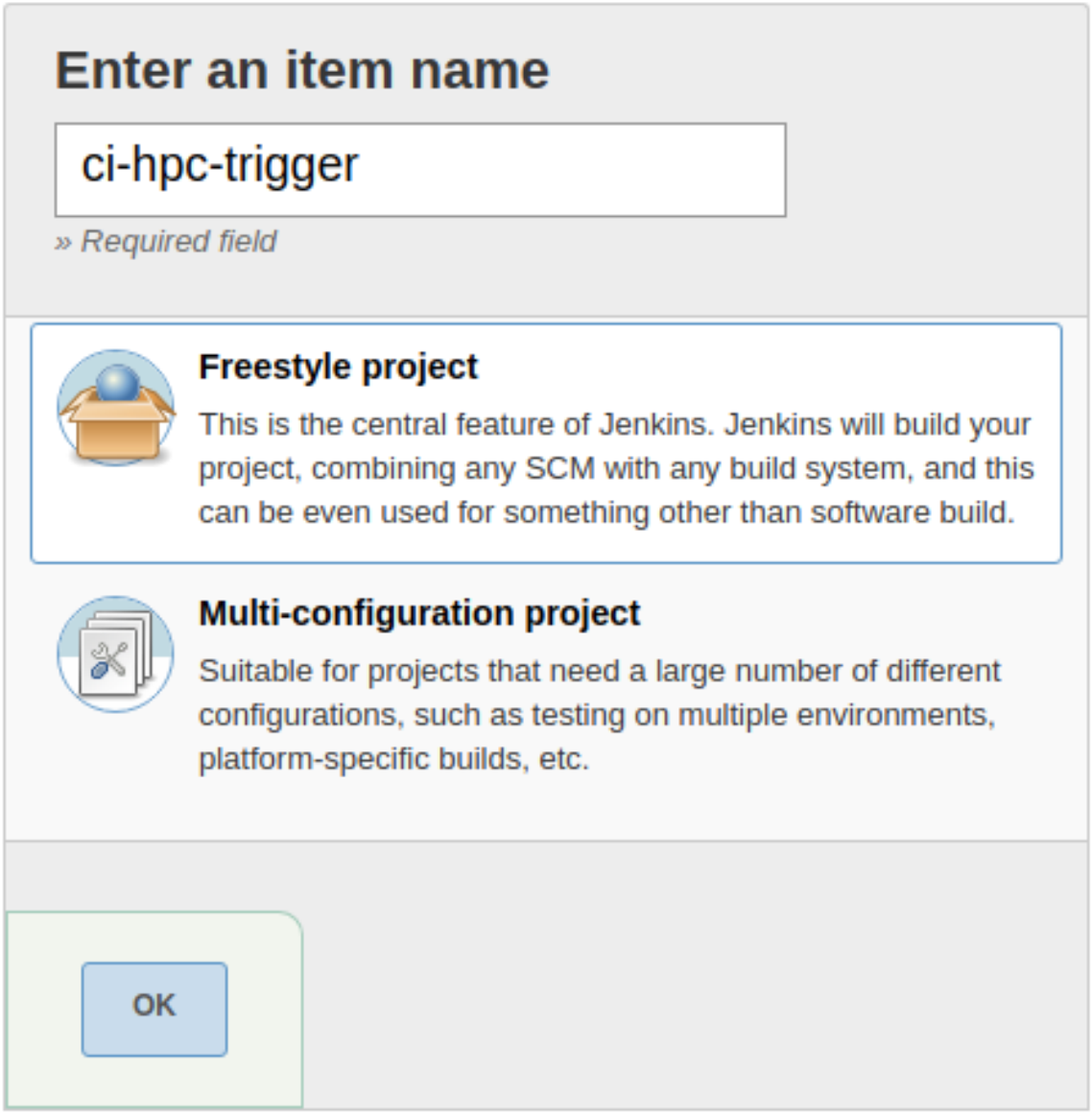
   Hopefully you should see something like this in your browser:

2. Configure Jenkins installation. No need to install all the plugins, but make sure you have the following plugins installed:
   (a) `Git`
   (b) `Credentials Binding`
   (c) `Parameterized Trigger`

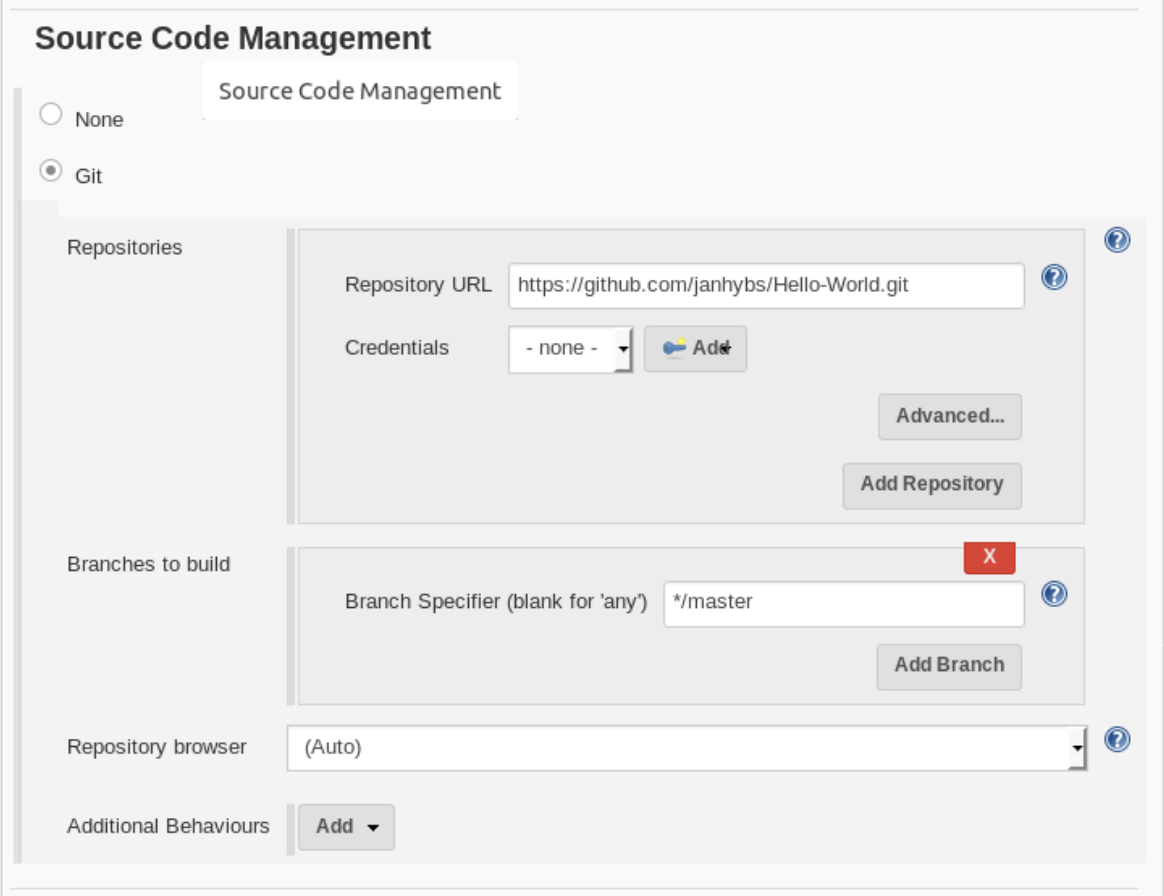


3.1 Configuring a Jenkins server

1. Create new Job (type can be `Freestyle project`)

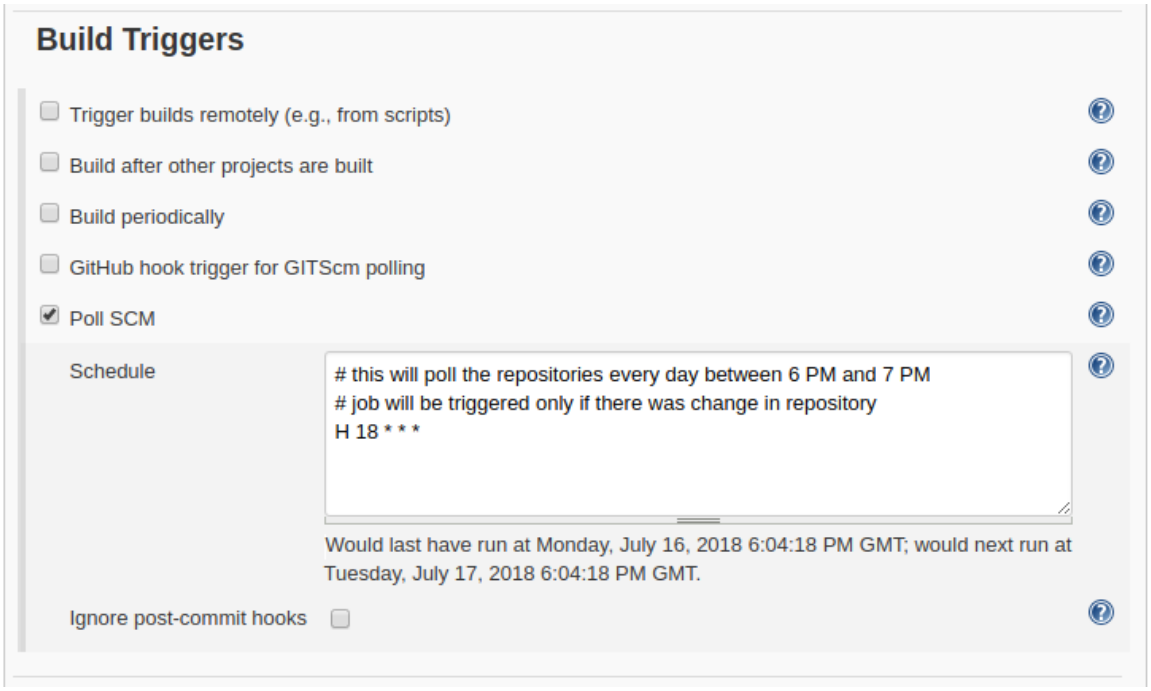


2. Setup git connection to your repository



3. Make sure `Poll SCM` is set (CRON syntax)

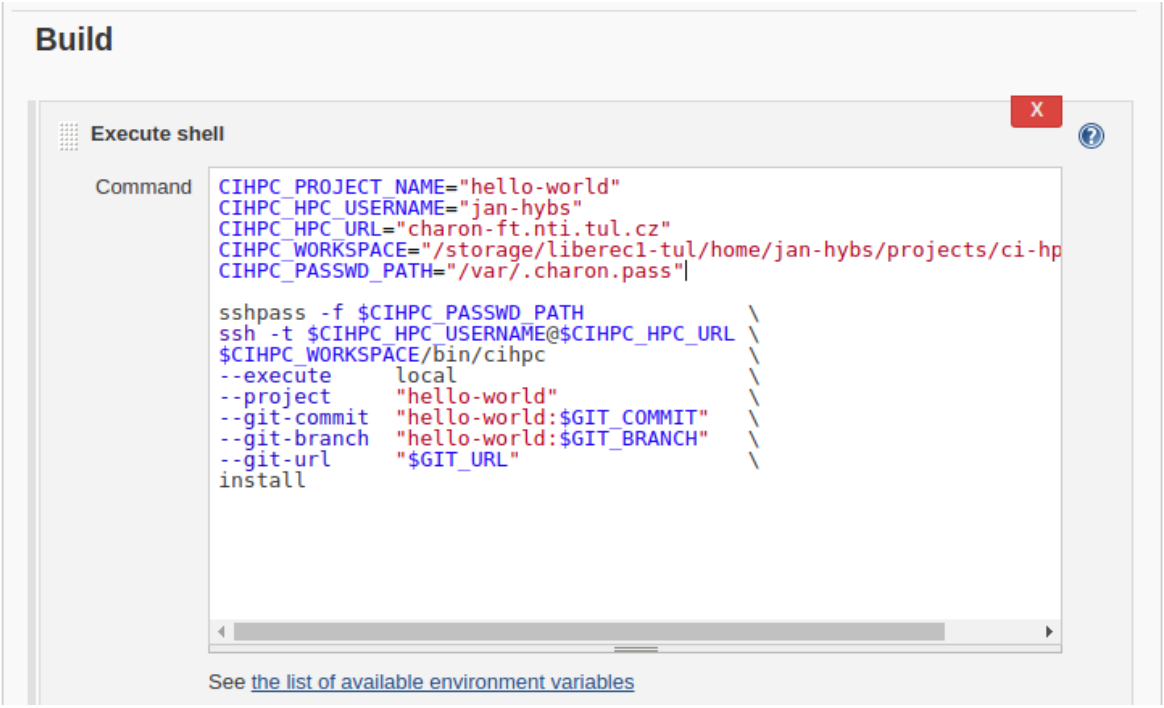
   Note: The value `H 18 * * *` translates to daily, between 6PM to 7PM



4. Add Shell `Build` step to your project:

   You can write your own script for starting cihpc but the most common scenarios are listed here. At the start of the `Build` step include your. project configuration:

```
CIHPC_PROJECT_NAME="hello-world"
CIHPC_HPC_USERNAME="jan-hybs"
CIHPC_HPC_URL="charon-ft.nti.tul.cz"
CIHPC_WORKSPACE="/storage/praha1/home/jan-hybs/projects/ci-hpc"
```



   - when using *SSH Key-Based Authentication*: Setup key-based SSH login to be able to login to an HPC server without password or ANY other prompts.

```
mkdir -p ~/.ssh
ssh-keyscan -H $CIHPC_HPC_URL > ~/.ssh/known_hosts

ssh -t $CIHPC_HPC_USERNAME@$CIHPC_HPC_URL      \
  $CIHPC_WORKSPACE/bin/cihpc                   \
     --project      "hello-world"              \
     --git-commit   "hello-world:$GIT_COMMIT"  \
     --git-branch   "hello-world:$GIT_BRANCH"  \
     --git-url      "$GIT_URL"                 \
     --execute      local                      \
     install
```

     Make sure you connect to the server at least once or automatically add entry to the known_hosts using commands:

```
mkdir -p ~/.ssh
ssh-keyscan -H $CIHPC_HPC_URL > ~/.ssh/known_hosts
```

   - when using *Password authentication*: If your server does not support key-less login, you can use `sshpass` (but it is not recommended as you need to keep your raw password somewhere on the server).

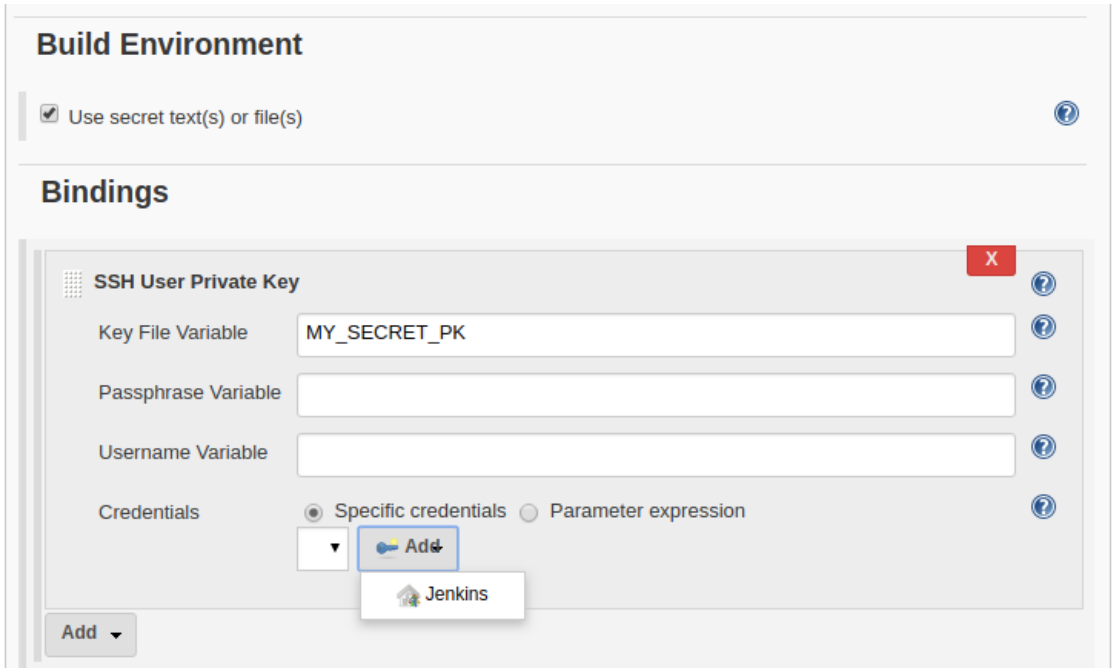
Note: make sure `sshpass` is installed on the Jenkins server and that your file containing password has permissions like `0400` (read only for owner).

```
mkdir -p ~/.ssh
ssh-keyscan -H $CIHPC_HPC_URL > ~/.ssh/known_hosts

CIHPC_PASSWD_PATH="/path/to/your/password-file"
sshpass -f $CIHPC_PASSWD_PATH                       \
   ssh -t $CIHPC_HPC_USERNAME@$CIHPC_HPC_URL        \
     $CIHPC_WORKSPACE/bin/cihpc                     \
        --execute     local                         \
        --project     "hello-world"                 \
        --git-commit  "hello-world:$GIT_COMMIT"     \
        --git-branch  "hello-world:$GIT_BRANCH"     \
        --git-url     "$GIT_URL"                    \
        --execute     local                         \
        install
```

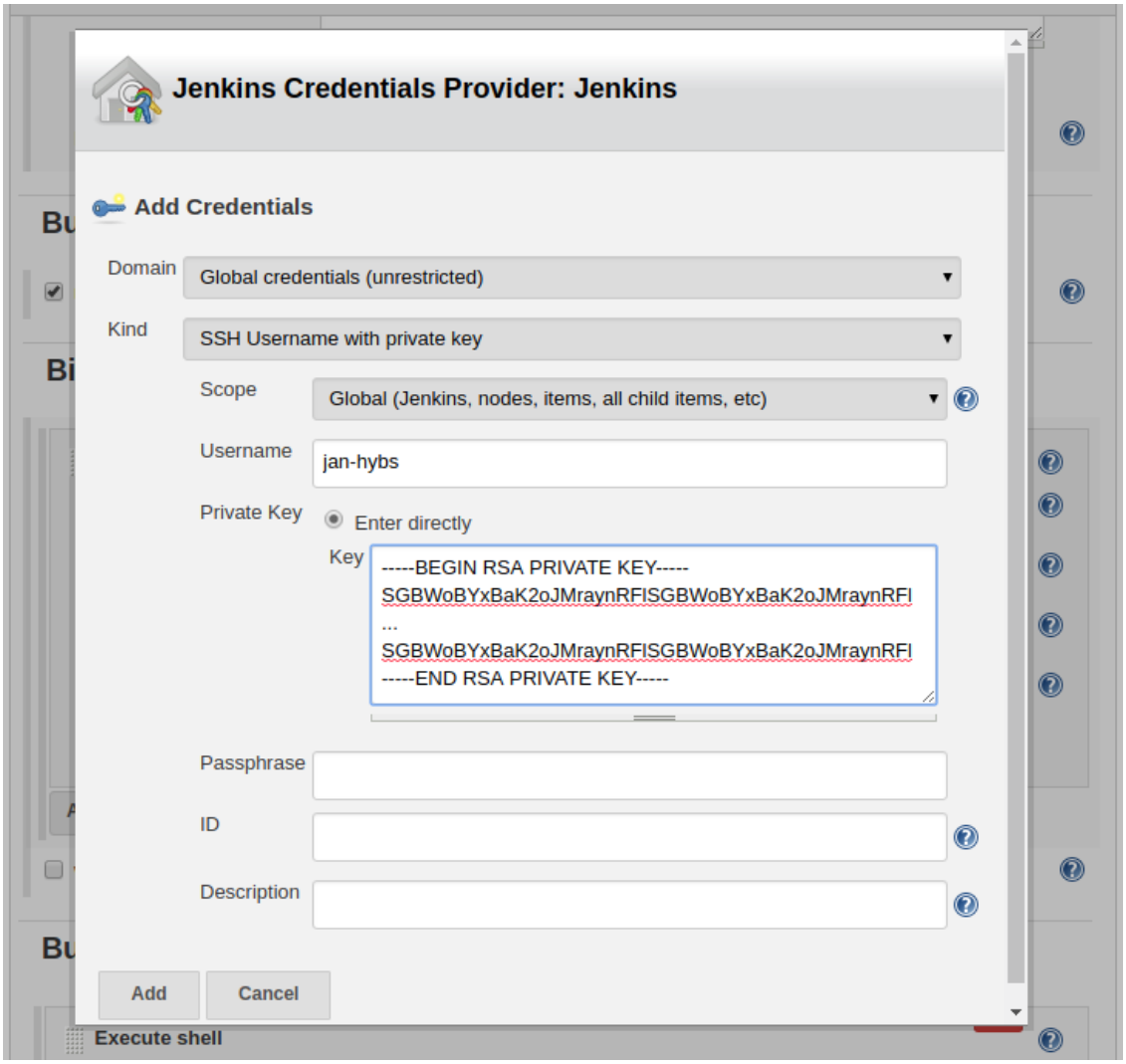
- when using *SSH Key-Based Authentication* (if you do not have access to the file system on Jenkins server, use `Credentials Binding` plugin)

```
CIHPC_PASSWD_PATH="/path/to/your/password-file"

mkdir -p ~/.ssh
cp $MY_SECRET_PK ~/.ssh/id_rsa
ssh-keygen -y -f ~/.ssh/id_rsa > ~/.ssh/id_rsa.pub
ssh-keyscan -H $CIHPC_HPC_URL > ~/.ssh/known_hosts

 ssh -t $CIHPC_HPC_USERNAME@$CIHPC_HPC_URL      \
   $CIHPC_WORKSPACE/bin/cihpc                   \
      --project     "hello-world"               \
      --git-commit  "hello-world:$GIT_COMMIT"   \
      --git-branch  "hello-world:$GIT_BRANCH"   \
      --git-url     "$GIT_URL"                  \
      --execute     local                       \
      install
```

Setup `Bindings`



Add `SSH Username with private key` kind



3.2 cihpc arguments

When calling `bin/cihpc` binary you can pass plenty of arguments (see the file), but couple of them are worth mentioning in this section:

- arguments `install` and `test`

  If `install` is given, will run all the `steps` within the `install` section. If `test` is given, will run all the `steps` within the `test` section.

  Example:

```
$> bin/cihpc --project=foo --execute local install
...
processing project foo, section ['install']
...
```

```
$> bin/cihpc --project=foo --execute local
...
```

(continues on next page)

(continued from previous page)

```
processing project foo, section ['install', 'test']
...
```

Note: by default both `install` and `test` are set, meaning entire project is processed.

- option `--execute`

  Valid values *for now* are either `local` or `pbs`. If set to `local`, the script will execute given section(s) on a **login node**. This can be usefull when **installing** your software. Example:

```
$> bin/cihpc --project=foo --execute local
executing script tmp.entrypoint-1532530773-2c4e85.sh using local system
...
```

  Note: by default no system is set, you should **always** set which system to use.

CHAPTER 4

Configuring project on HPC server

Note: Assuming we are testing project named `hello-world`.

1. Login to HPC server and clone `ci-hpc` repository:

```
cd $WORKSPACE # directory where you keep your projects
git clone https://github.com/janhybs/ci-hpc.git
cd ci-hpc
```

1. Install necessary pip packages:

   Execute `install.sh` script located in the `bin` folder. It is basically shortcut for a `pip3 install -r requirements.txt`. You can also pass any arguments to the pip. **It is expected to have `python3` and `pip3` in the path.**

   ```
   bin/install.sh --user --upgrade
   ```

   To install packages system wide, do not add the `--user` flag:

   ```
   bin/install.sh --upgrade
   ```

2. Create configuration file `config.yaml` for the project

```
export PROJECT_NAME=hello-world

mkdir -p cfg/$PROJECT_NAME
nano     cfg/$PROJECT_NAME/config.yaml
```

1. Setup `config.yaml` configuration file

Please refer to config.yaml section to find out more about configuration.

CHAPTER 5

config.yaml specification

5.1 Terminology

- `section`

  By a section we understand either `testing` or `installing` section. A section is a main configuration block which groups together installation or benchmark testing procedures. A `section` can contain zero or more `steps`.

  Note: Testing section named `install` should contain a configuration for the project installation, compilation or even git cloning. Testing section called `test` contains configuration for the benchmark testing.

- `step`

  A step is main unit which can contain `shell` commands, git cloning and more.

- `shell`

  A shell part can contain `bash` commands (multiline line string)

5.2 config.yaml example

```
# start of a install section
install:

  # first step in the install section
  - name: repository-checkout
    git:
      - url: git@github.com:janhybs/bench-stat.git

    # you can also omit shell if there is no need for it
    shell: |
      echo "By this point, the repository is already cloned"
      echo "And checkout out to latest commit"
```

(continues on next page)

(continued from previous page)

```
  # seconds step in the install section
  - name: compilation-phase
    shell: |
      cd bench-stat
      ./configure --prefix=$(pwd)/build
      make && make install

# start of a test section
test:
  # first step in the test section
  - name: testing-phase
    shell: |
      cd bench-stat
      build/O3.out
```

5.3 config.yaml structure

Note: keys in `[brackets]` are optional.

```
# value is list of steps
install:
    # name of the step
  - name: string

    # description of the step
    [description]: string

    # default true, if true step is enabled
    # will be processed, otherwised will be skipped
    [enabled]: boolean

    # default false, if true shell is started with set-x
    [verbose]: boolean

    # default 1, number of this step repetition
    # (useful for benchmark testing)
    [repeat]: int

    # bash commands to be executed
    [shell]: string

    # default log+stdout, how should be output
    # of the shell be displayed, possible values:
    #   log         - logging to log file only
    #   stdout      - only display output
    #   log+stdout  - combination of both
    [output]: string

    # if set, will execute shell in side container
    # value must command(s) which when called will
    # start container (docker/singularity), command
    # must contain string %s at the end
    # %s will be subsituted with a suitable command
```

(continues on next page)

(continued from previous page)

```
    #
    # examples:
    #   container: |
    #     docker run --rm -v $(pwd):$(pwd) -w $(pwd) ubuntu %s
    #   container: |
    #     module load singularity
    #     singularity exec -B /mnt sin.simg %s
    [container]: string

    # complex type, if set, will create build matrix of variables
    # detailed explanation below
    [variables]: <variables>

    # complex type, if set, will collect results
    # detailed explanation below
    [collect]: <collect>
```

5.3.1 config.yaml `variables` specification

This field will allow you to create so called `build matrix` of all possible combinations of the given variables. It is especially useful when running multiple benchmarks which are basically the same and only thing which is different are arguments passed to the binary. It this case there is no need to copy the `step` in the install section over and over again only to change a single word in `shell`. The principle is the same as the `build matrix` used in a `.travis.yml`

You can specify this fields and you can set unlimited amount of variables and their values like this:

```
variables:
  - matrix:
      - var-name: [value-1, value-2, value-3]
      - foobar:   [1, 2, 3, 4]
      - test:     [cache, io]
```

The exmaple above will expand to 24 (`3 * 4 * 2`) individual configurations, variables are available in the `shell` field (and in the `extra` field of a `collect` field).

A shell `field` can use these variables with help of placeholders which are in `<variable>` form, usage like this:

```
    shell: |
      echo "Running test <test> with arguments foobar=<foobar> and var-name=<var-name>
↪"

      # the first echo will look like this:
      # Running test cache with arguments foobar=1 and var-name=value-1

      benchmark/O3.out <test> --foobar=<foobar> -v <var-name>
      # the first call of the binary benchmark/O3.out will look like this:
      # benchmark/O3.out cache --foobar=1 -v value-1
```

The value of the variable in a `matrix` field must be an array. It can be array of strings, ints, floats, or even **dictionaries**. The example below will demonstrate usage of dictionaries.

```
variables:
  - matrix:
      - foobar:
          - foo:  10.65    # the first value
```

(continues on next page)

(continued from previous page)

```
            bar:  -3.14
          - foo:   1.05   # the second value
            bar:  42.00
      - test: [cache, io]
```

and usage in shell:

```
    shell: |
      echo "Running test <test> with arguments foo=<foobar.foo> and bar=<foobar.bar>"

      # the first echo will look like this:
      # Running test cache with arguments foo=10.65 and bar=-3.14"

      benchmark/O3.out <test> --foo=<foobar> --bar=<var-name>
      # the first call of the binary benchmark/O3.out will look like this:
      # benchmark/O3.out cache --foo=10.65 --bar=-3.14
```

There can be multiple `matrix` fields as well (for when you don't want all the combinations):

```
variables:
  - matrix:
    - benchmark: 01_square_regular_grid
    - mesh:
      - 1_15662_el
      - 2_31498_el
      - 4_62302_el
      - 8_124498_el
  - matrix:
    - benchmark: 02_cube_123d
    - mesh:
      - 1_15786_el
      - 2_29365_el
      - 3_47367_el
      - 4_58803_el
```

5.3.2 config.yaml `collect` specification

If you specify `collect` in a step of the install section, CI-HPC framework will automatically look for the benchmark results in form of `json` or `yaml` files. But you have to tell CI-HPC, what these files are, and how to work with them.

```
collect:
  # value  must be a string containing a path specification.
  # pathname can be either absolute (like /foo/bar/result.json) or
  # relative (like bar/*/*.json), and can contain shell-style wildcards
  # double asterisk ** will match any files and zero or
  # more directories and subdirectories
  # the value is usually something like
  # directory/*.json
  # more here https://docs.python.org/3.6/library/glob.html
  files: string

  # path to the repository from which git information is taken
  # if set will determine commit, branch and datetime of the current HEAD
  repo: string
```

(continues on next page)

(continued from previous page)

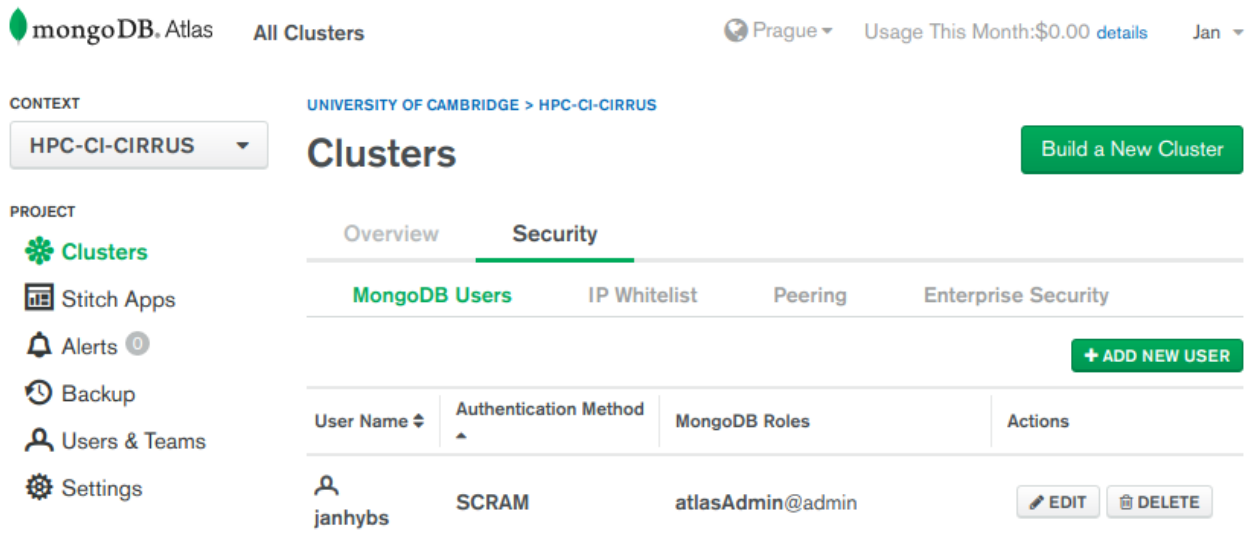
```
# a path to the python module which will take care of the parsing and
# storing. There is a generic module which does a decent job, so if
# your result output format can be easily edited, it will work just fine
[module]: string

# location where matched files can be moved to after the files has
# been processed. This will simply move the files to a location
# so if you have multiple files from single execution with the same name
# they will be overwritten, to avoid that see 'cut-prefix' below
# You can avoiding processing the same results twice.
# (it is recommended to put the files away)
[move-to]: string

# if move-to is set, will cut the path prefix of your files
# it is especially useful when your results are located deep structure
# or if they are in a structure, you want to preserve
[cut-prefix]: string

# after the processing is done, you can add some extra properties on top
# you can even use variables from build matrix.
# for example:
#     extra:
#       foo: true
#       size: <test-size|i>
# will put extra two fields to a document,
# which is headed to the database
# they will be put in a system field:
#     {
#       system: {
#         foo: true,
#         size: 1024,
#         ...
#       },
#       problem: {
#         ...
#       }
#     }
# The second variable size will be that of the typo of integer
# this is bacause |i was specified at the end. All possible
# conversions are:
#     |s for string (default)
#     |i for integer
#     |f for floats
[extra]: dictionary

# if true, will save the processed results to the DB
[save-to-db]: boolean
```

CHAPTER 6

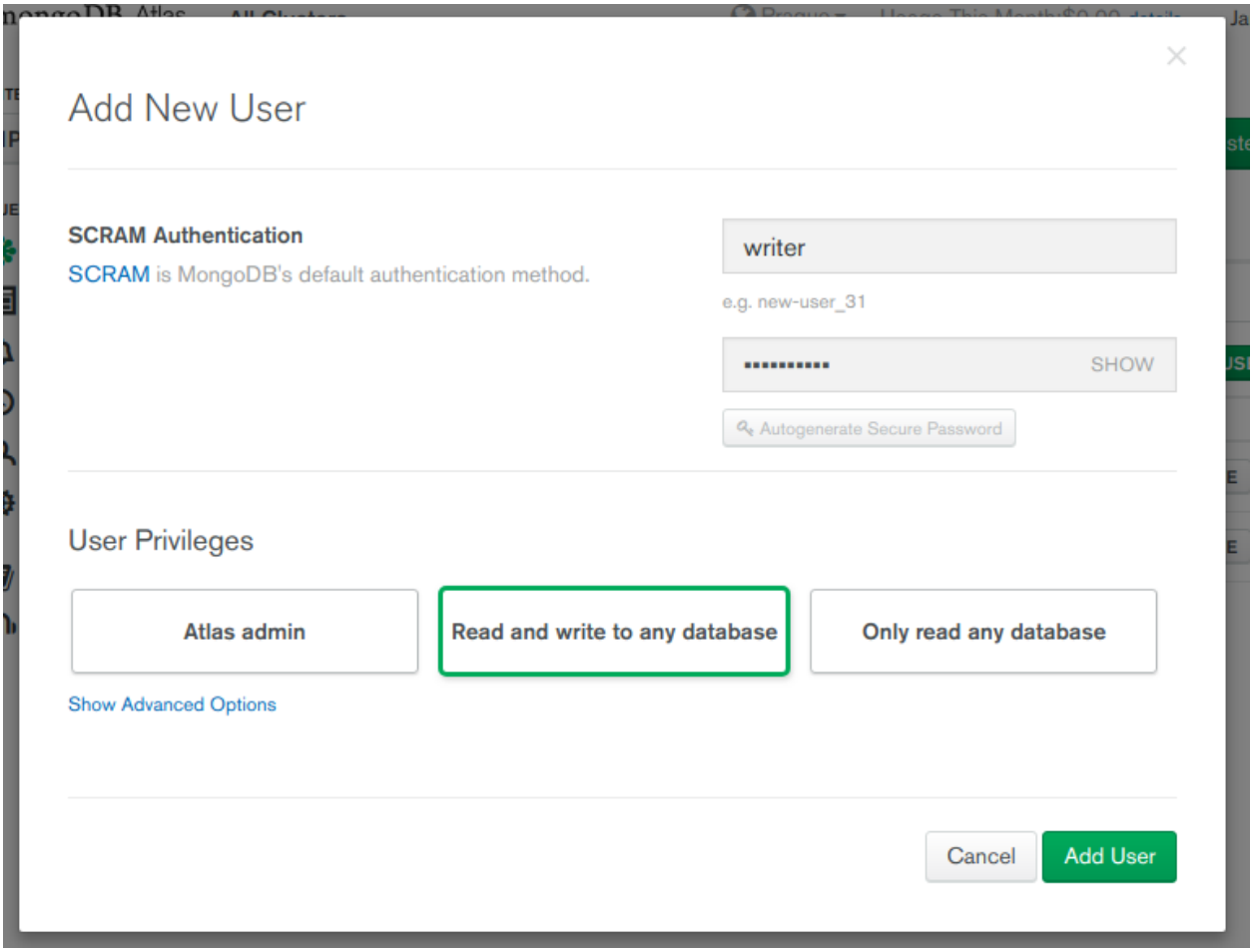
MongoDB configuration

Configuring MongoDB storage is basically just creating an user, which has permissions to read and write to a database.

When using MongoDB atlas you setup your project. By default you should have (or you should be asked to add) a user, which is in role of a admin.



It is recommended to add another user, who can write to any database:



After the user is created you need to create a file `secret.yaml` inside `cfg` directory. **Make sure only owner can read this file** as it will contain username, password and server to the MongoDB database.

6.1 `secret.yaml` structure

The file can contain configuration for multiple project, the main section is same as the name of your project (e.g. `hello-world`). To setup connection to a db server create section `database`This section can contain several options but all of them are passed to the construcotr of the python's pymongo.mongo_client.MongoClient constructor. Please refer to api.mongodb.com for further information.

Take a look at the example of the `secret.yaml` file here.

If your MongoDB server is not hosted, you must setup MongoDB authorization (for example via `/etc/mongodb.conf`):

```
# /etc/mongodb.conf
net:
  bindIp: 0.0.0.0
  port:   27017

security:
  authorization: enabled
```

6.1.1 `secret.yaml` examples

1. Example 1 (single host):

```
hello-world:
  database:
    host:           [mongodb.server.example.com:27017]
    connect:        true
    authSource:     admin
    username:       writer
    password:       password-here
```

2. Example 2 (mongodb configuration with 3 hosts):

```
hello-world:
  database:
    host:
      - cluster0-shard-00-00-foobar.mongodb.net:27017
      - cluster0-shard-00-01-foobar.mongodb.net:27017
      - cluster0-shard-00-02-foobar.mongodb.net:27017
    replicaSet:     Cluster0-shard-0
    connect:        true
    authSource:     admin
    authMechanism:  SCRAM-SHA-1
    ssl:            true
    username:       writer
    password:       password-here
```

If `secret.yaml` is setup properly, you can easily collect benchmark data to a database. By default the data will be saved into a database with the name of your project name. You can change this behaviour by adding another section to the `secret.yaml`:

```
hello-world:
  artifacts:
    db_name:          customDbName      # name of the database to save date to

    col_timers_name:  customRepColName  # name of the collection which will contain
                                        # benchmark data (reports, frames)

    col_files_name:   customFSColName   # name of the collection which will contain
                                        # files and logs when error during run occured
```

Note: Assuming we are trying to override artifacts location fot the project `hello-world`

CHAPTER 7

Configuring Flask server

Note: assuming you have an Apache working and running.

7.1 Start the flask server with the help of a `bin/server` script:

```
bin/server start
```

To test the server is running, execute:

```
bin/server status
```

And to stop the running server call:

```
bin/server stop
```

And if you visit the url `http://0.0.0.0:5000/` in your browser (it may take couple seconds), you shoud see the message:

```
Your server is running!
```

In opposite case, check the log `ci-hpc.log` located at the repository root or try to to execute the script `bin/server` without any arguments (this will start the server **not** in the background)

7.1.1 Configuring the server host and port

By default the server is accessible for anyone. You can restrict this by scecifying `--host=<hostmask>` where `<hostmask>` is the hostname to listen on. Default to `0.0.0.0`.

To change the port of the server API server specify `--port=<portval>` options, where `<portval>` is the interger value of your **API** server port.

To see all the options you can change see `bin/server -- --help`.

7.2 Configuring www folder

Edit `index.html` located in `www` folder. Lines `41` and `42` is all you need to change. Simple change the values so they match your project and server:

By default project is set to `hello-world` and ip is just a dummy url. The IP you specify must be accessible by another computer!

```
projectName: 'hello-world',
flaskApiUrl: 'http://flask.server.example.com:5000',
```

7.3 Visualisation settings aka what to visualise

Edit visualisation settings for yout project The yaml file is located at `cfg/<project>.yaml`. e.g. if you have project with name `foo`, the location is `cfg/foo.yaml`

This configuration is reasonably straightforward. You fill out the info about your project and then just say what variables will be used for what cause. Take a look at example file which explains what variable is for cause.

7.4 [optional] Create a symlink to Apache www folder:

Note: assuming you are located at the repository root

```
ln -s $(pwd)/www /var/www/html/ci-hpc
```

If you visit http://localhost/ci-hpc you should see the the results.