Manual Molgenis/Compute

Genomics Coordination Center

December 14, 2012

Contents

1 Introduction

This manual explains how one can use *Compute* to generate a workflow of analysis tasks (c.q. scripts) that can be executed on a distributed system.

To use *Compute*, you need to create the following.

- A *workflow* that describes a series of steps and the order in which they should be executed. Each step refers to a protocol and eventually results in a (e.g. bash or R) script.
- A set of *protocols*, where each protocol is a template in the Freemarker language that may contain parameters. Each protocol applies to a certain target type.
- A *parameter list* which describes all parameters that are used in the protocols.
- A set of *targets* on which the protocols are applied.

These four inputs may be stored either in files or in a database. Likewise, *Compute* can store its generated output in files or in a database, too.

First, section 2 ”A Hello World workflow” explains the basic use of *Compute*. Next, section 3 ”Extending our Hello World workflow” extends that workflow and explains more advanced features. Both sections assume that you run *Compute* from the command line. Alternatively, section **??** ”**??**” shows how one can run *Compute* from a database.

2 A Hello World workflow

This section explains how one can use *Compute* to generate some shell scripts that print invitations for a party to the standard out. In addition, *Compute* generates a couple of other shell scripts that can be used to execute these two scripts on a distributed system or on one’s local machine. Section 2.6 ”Generated scripts” discusses them in detail. The following four subsections below explain how one can create a workflow, a protocol, a parameters list and a set of targets, to generate the Hello World workflow.

2.1 A workflow with only one step

The most simple workflow would contain only one step. Let's call the step `GuestInvitationStep`. Now create a file called *workflow.csv* with the following content:

```
name,                    protocol_name,     PreviousSteps_name
GuestInvitationStep,     GuestInvitation,
```

The first row in this file contains the column headers. Each of the following rows describes a step in the workflow. In the first column you'll find the step's name (unique per workflow), followed, in the second column, by the protocol's name (without its extension `.ftl`) that you want to run in this step. The third column contains a comma separated list of step names that should be finished before the step in a row starts. So, the third column refers to the first column, and not to the second column. In our current example, the list in the third column is empty.

2.2 A protocol to invite guests

A protocol generally is a template of a shell script in the Freemarker[1] language that describes the work to be done.

Let's now create a directory called *protocols* and save our first protocol file *GuestInvitation.ftl* in there, with the following content.

```
echo "Hello ${guest},"
echo "We invite you for our ${party}."
```

A protocol may contain one or more parameters, like `guest` and `party` in our example. The idea is that you can generate multiple shell scripts, given different values for these parameters. Given a value for each of the parameters, this protocol echos an invitation to the standard out.

2.3 Parameters: constants and variables

Each parameter that is used in a protocol should be defined in a parameters file. Let's create such a file and call it *parameters.csv*. Now add our two parameters `guest` and `party` as follows to this file.
There are two types of parameters. First, parameters may be constants, like our parameter `party` which has a default value `wedding`, as shown above here.

[1]See http://freemarker.org/ for a manual.

```
name,   defaultValue,  description,  dataType,  hasOne_name
guest,  ,              ,             ,
party,  wedding,       ,             ,
```

So, in each of the generated scripts, the value of the parameter `party` will be `wedding`. If you want to send the same invitation, but for a different party, you only have to change this value in one place. Second, parameters may be variables and have a different value in each of the scripts that are generated from a given protocol. The next section will explain how these parameters, like `guest` in our case, and their different values are defined in a worksheet.

2.4 A worksheet with guests

Let's now create a worksheet, save it as *worksheet.csv*, and add the following content.

```
guest
Charly
Cindy
Abel
Adam
Adri
```

The idea of the worksheet is as follows. The first row contains parameter names (c.q. target types), comma separated. In our case we only have one parameter, called `guest`. Each of the following rows is called a *target*. When running *Compute*, the protocol above is subsequently applied to each of the targets. So, in our example, we will generate a different invitation for each of our guests.

2.5 Running *Compute*

You need at least two command line parameters to run compute: `input` and `id`. The first parameter (`input`) refers to the directory in which you have stored your *workflow.csv*, *protocol* directory, *parameters.csv* and *worksheet.csv*. Alternatively, you may specify each of these parameters individually by `-workflow`, `-protocols`, `-parameters`, `-worksheet`, and `-scripts`. Where the parameter `scripts` refers to the directory where *Compute* will store the generated scripts. Its default value is equal to the value that you assign to the `id` parameter. In your protocols, you may want to use the values of the command line parameters. However, be aware that the parameter names you have to use are slightly different from the command line parameters: `${McWorkflow}`, `${McProtocols}`, `${McParameters}`, `${McWorksheet}`, and `${McScripts}`.

The second command line parameter (`id`) may have a different value, every time you generate. This may be useful, for example, in case you want to redo an analysis after slightly changing a protocol and compare the outcomes of both analyses. You could do so by making the `McId` parameter part of the file or directory names in which you store your output. In this way, you won't overwrite the output of the first analysis so that you can compare it with the outcome of our your second analysis.

Let's now generate the scripts with the invitations by running the following command. We assume that you have put your workflow, parameters, worksheet files and protocols directory in a directory called `helloWorld`.

```
sh molgenis_compute.sh \
-input=helloWorld \
-id=run01
```

2.6 Generated scripts

So, how do the generated scripts in the `scripts` directory look like? Let's first consider one of the five scripts that contain the invitations to our five guests: `run01_s00_GuestInvitation_1.sh`. The script name is constructed as follows.

- `run01`: the id that you used when you ran *Compute*.
- `s00`: the step number in your workflow, starting from zero.
- `GuestInvitation`: the corresponding workflow step name.
- `1`: the line number in the worksheet.

The step number, step name and line number are separated by underscores. Let's open the script and view its content:

```
echo "Hello Charly,"
echo "We invite you for our wedding."
```

Now let's open the second script, `run01_s00_GuestInvitation_2.sh`:

```
echo "Hello Cindy,"
echo "We invite you for our wedding."
```

These scripts correspond to the `GuestInvitation` protocol (section 2.2), where `${party}` got the constant value ”wedding”, and `${guest}` got a different value each time, as defined in the worksheet. Because ”Charly” is the first target in the worksheet, she ends up in the first script. Correspondingly, because ”Cindy” is the second target in the worksheet, she ends up in the second script. And so on.

Next to these two analysis scripts, three submit scripts are generated.

- *runlocal.sh* executes the analysis scripts sequentially.
- *submit.sh* submits the analysis scripts to a PBS scheduler for parallel execution where possible.
- *submitCustom.sh* is a script that is based on the protocol `CustomSubmit.sh` which you can find in the protocols directory. You can customize the way the analysis scripts are submitted by customizing this protocol.

You can use these to submit and start the execution of your analysis scripts in the right order. Section 3.1 ”A workflow with dependencies” below will explain how you can define the order between the steps in your workflow. After executing the analysis scripts, two invitations will be echo'ed to the standard out.

In addition, a copy of your workflow, parameters and worksheet file are put in the `scripts` directory, as well.

3 Extending our Hello World workflow

This section adds more complexity to the ”Hello World” workflow we've developed above and demonstrates that *Compute* can generate more sophisticated workflows, too. In addition of only inviting guests to our wedding, we will also organize some activity for our guests. The guests will be divided in two groups: child or adult. Each group has one organizer that will plan an activity for his group. After sending out the individual invitations to our guests, for each group, we will send its organizer a letter with a guest list.

3.1 A workflow with dependencies

Let's call the step that sends a letter to each organizer `OrganizerInvitation`. Suppose that before starting this step, we want the `GuestInvitation` step to be finished first. Let's add the new step to our *workflow.csv* file and define its dependency on the `GuestInvitationStep` step.

```
name,                    protocol_name,          PreviousSteps_name
GuestInvitationStep,     GuestInvitation,
OrganizerInvitation,     OrganizerInvitation,    GuestInvitationStep
```

Adding `GuestInvitationStep` to its *PreviousSteps_name* will ensure that the `GuestInvitation` scripts will be finished before the `OrganizerInvitation` step will be started. Be aware that the values in the third column refer to those in the first column, and not to those in the second column.

3.2 A new protocol for each group

Let's create a new protocol and save it as *OrganizerInvitation.ftl* in the protocols directory and add the following content.

```
#FOREACH group

echo "Dear ${organizer},"
echo "Please organize activities for the ${group} group."
echo "List of guests:"
<#list guest as g>
    echo "${g}"
</#list>
```

This paragraph will explain this protocol in detail. In the protocol, we introduce a new parameter `group` which may have the values `child` and `adult`. We will specify these values in the worksheet in section 3.4.1, below. The section 1 "Introduction" already mentioned that each protocol applies to a certain target type. The `#FOREACH group` statement in the first line of this protocol means that this protocol will be applied to each different value that `group` has in the worksheet; i.e., it will be applied once to `child`, and once to `adult`. What happens under the hood, is that the worksheet is *folded* based on the specified target. The folding reduces the worksheet to only two lines, one for each group. This will thus result in a list of guests per group. This protocol iterates through that list of guests by making use of the `<#list>` Freemarker syntax. Section 3.5 "Folding the worksheet in the OrganizerInvitation step" explains the folding of the worksheet as a result of the `#FOREACH group` statement, in detail. That section also explains why the parameter `organizer`, which is also new in this protocol, can also be used as a value, instead of a list.

3.3 Parameters with interrelationships

So, we need to define the new parameters `group` and `organizer`. Let's add them to the *parameters.csv* file as follows.

```
name,        defaultValue,  description,  dataType,  hasOne_name
guest,       ,              ,             ,
party,       wedding,       ,             ,
organizer,   ,              ,             ,
group,       ,              ,             ,          organizer
```

We have also added relationships between group and organizer in the `hasOne_name` column: a group has only one organizer. We'll come back to the exact meaning of this relationship in section 3.5 "Folding the worksheet in the OrganizerInvitation step".

3.4 Extending the worksheet

In the first subsection below we will extend the worksheet with our new parameters. Section 2.4 "A worksheet with guests" already explained that because the `GuestInvitation` protocol does not contain a `FOREACH` statement, it is subsequently applied to each of the targets (i.e. each line) in the worksheet.

If a protocol does contains the `FOREACH` statement, the worksheet will however first be folded. The second subsection below will explain the folding process.

3.4.1 The worksheet in GuestInvitation step

Let's take the worksheet from section 2.4 "A worksheet with guests" add a `group` (child or adult) to each child. Let's also add an organizer to each group and update the *worksheet.csv* file as follows.

```
guest,    group,  organizer
Charly,   child,  Oscar
Cindy,    child,  Oscar
Abel,     adult,  Otto
Adam,     adult,  Otto
Adri,     adult,  Otto
```

3.5 Folding the worksheet in the OrganizerInvitation step

In principle, a protocol is applied to each of the targets in the original worksheet. However, if a protocol contains a `#FOREACH` statement, then the worksheet will first be *folded*. After the folding, the protocol will be applied to each line in the folded worksheet. Because the `OrganizerInvitation` protocol starts with "`#FOREACH group`", it will be executed *for each* different value of group (i.e. child and adult). Under the hood, this boils down to folding the original worksheet based on group. After folding, each line contains a different value of group:

```
guest,                 group,  organizer
[Charly, Cindy],       child,  Oscar
[Abel, Adam, Adri],    adult,  Otto
```

For each group, you'll get a list of guests which are indicated with the brackets [and]. Section 3.2 "A new protocol for each group" above shows how a protocol can iterate through such a list.

So, folding on a certain target, results in lists of the other targets. However, although we do see a list of guests, we don't see a list of organizers. The reason for this is that in the parameter list in section 3.3 "Parameters with interrelationships" we have specified that group *has one* organizer. Consequently, for each value of group we have only one value for organizer, too. I.e. each list of organizers is reduced to only one single value.

Obviously, because we stated that one group can only have one organizer, it would not make sense to have a list of organizers with identical elements for each of the groups.

3.6 Advanced features of a protocol

The following subsections explain advanced features of a protocol.

3.6.1 Adding a header and footer to each protocol

In your protocols directory you may create the two files "Header.ftl" and "Footer.ftl". If present, these files will be respectively prepended and appended to each of your protocols when generating a workflow.

3.6.2 Hardware requirements specification

The header of a protocol may contain the following line in which you specify the hardware requirements for your analysis.

```
#MOLGENIS walltime=hh:mm:ss mem=m nodes=n cores=c
```

where walltime is the maximum execution time, m is the memory (*e.g.* 512MB or 4GB), n is the number of nodes and c is the number of cores that you request for the execution of this analysis.

3.6.3 Software tools

In your protocols you may want to make use of some software tools that are already installed on the backend where your scripts will be run. However, the path to these tools may vary between different backends. One solution to this is to put the tools in the $\$PATH$, so that you can just call them without specifying the path. On two backends, i.e. `cluster.gcc.rug.nl` and `grid.sara.nl`, we made it quite easy for you to do so. The following statement will add a tool, say $yourModule$, to the path.

```
module load yourModule
```

On the two backends, the following modules are available so far:

- `bwa/0.5.8c_patched`
- `capturing_kits/SureSelect_All_Exon_30MB_V2`
- `capturing_kits/SureSelect_All_Exon_50MB`
- `capturing_kits/SureSelect_All_Exon_G3362`
- `fastqc/v0.7.0`
- `fastqc/v0.10.1`
- `gtool/v0.7.5_x86_64`

- `impute/v2.2.2_x86_64_static`
- `jdk/1.6.0_33`
- `picard-tools/1.61`
- `plink/1.07-x86_64`
- `Python/2.7.3`
- `R/2.14.2`

A protocol that wants use plink, for example, may look like this:

```
  module load plink/1.07-x86_64

  plink --noweb --bfile $WORKDIR/lspilot1/GvNL_good_samples.out4
--het --out $WORKDIR/lspilot1/GvNL_good_samples.out7
```

3.6.4 Get your files

A protocol may be executed in a distributed environment. As a result, the data may not be available on the node where the execution takes place. Therefore, one should first download the data to the execution node. In some distributed environments this may involve a series of statements that one actually does not want to care about. To make this process easier for our users, we come with the following solution. For every file that you want to use in the analysis protocol, you may include the following statement in the protocol before using it.

```
  getFile "$myInputFile"
```

Where `"myInputFile"` is a parameter in your parameter list that refers to the file. The `getFile` command will then take care of putting your data in the right place.

3.6.5 Put your files back

After finishing the analysis, you may save the files you want to keep by including the following statement at the end of your protocol.

```
  putFile "$myOutputFile"
```

Where `"myOutputFile"` again is a parameter in your parameter list that refers to the respective file. The `putFile` command takes care of all the work needed to store your data in the right place.

4 Deployment of the database version of *compute*

This part of the tutorial explains who to deploy the Database version of the Molgenis/compute and submit jobs to the grid (glite-wms) system using the ″pilot″ job approach.

4.1 Requirements

Molgenis/compute can be deployed and ready to submit jobs to the grid scheduler via `ssh` just in few straightforward steps. We prepared a shell scripts to automate every deployment and utilise step. The scripts are can be found in Molgenis github:

```
https://github.com/molgenis/molgenis_apps/tree/testing
/modules/compute4/deployment/
```

The next modules are required to be present in the system for Molgenis/compute deployment:

- java 1.6.0 or higher
- git 1.7.1 or higher
- ant 1.7.1 or higher
- mysql 5.1.54 or higher

4.2 Database creation

The ″compute″ database should be created in the MySQL server. Run the following commands for this:

```
   Start mysql.
   CREATE USER 'molgenis' IDENTIFIED BY 'molgenis';
   CREATE DATABASE compute;
   GRANT ALL PRIVILEGES ON compute.* TO 'molgenis'@'%'
WITH GRANT OPTION;
   FLUSH PRIVILEGES;
   Logout.
```

If you have further questions about database creation, please follow the MOLGENIS development manual at http://www.molgenis.org/wiki/MolgenisGuide,

4.3 Project check-out and build

Create the project root directory in your system and clone the molgenis project from the github repository with the following commands:

```
git clone https://github.com/molgenis/molgenis.git
git clone https://github.com/molgenis/molgenis_apps.git
cd molgenis_apps
ant -f build_compute.xml clean-generate-compile
```

Alternatively, you can use `clone_build.sh` for it.

4.4 Start the server

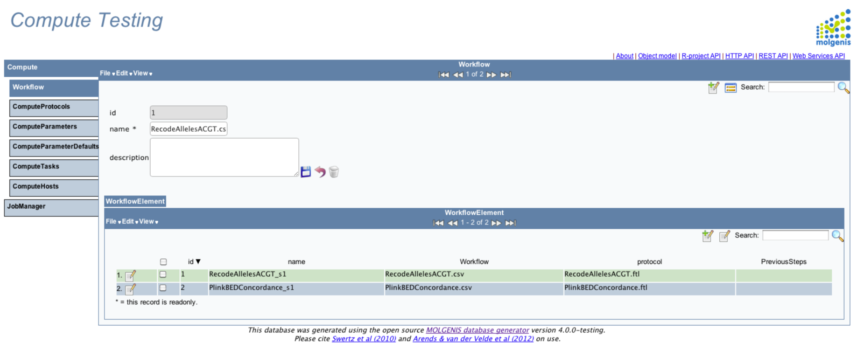
Now, the compute project is built and you can start the web-server and run the DB version of compute with running the following command:

```
nohup ant -f build_compute.xml runOn -Dport=$1 &
```

In your console, you should see output like:

```
****************************************************
APPLICATION IS RUNNING AT: http://localhost:8080/compute/
****************************************************
```

Later, you can copy the link into your browser and you will see generated user interface with empty database, like in Figure 1. In this example, the DB contains two workflows.



id: 1
name *: RecodeAllelesACGT.cs
description:

		id	name	Workflow	protocol	PreviousSteps
1.	☐	1	RecodeAllelesACGT_s1	RecodeAllelesACGT.csv	RecodeAllelesACGT.ftl	
2.	☐	2	PlinkBEDConcordance_s1	PlinkBEDConcordance.csv	PlinkBEDConcordance.ftl	

* = this record is readonly.

This database was generated using the open source MOLGENIS database generator version 4.0.0-testing. Please cite Swertz et al (2010) and Arends & van der Velde et al (2012) on use.

Figure 1: An example of Molgenis compute UI

Alternatively, you can use `sh restart.sh` script for it. Run the script specifying the port on which you like to run the web server

```
sh restart.sh <your_port>
```

4.5 Workflow import and execution task generation

You can use the `sh importWorkflow.sh` script to import a workflow into a database. Run it with few parameters:

```
sh importWorkflow.sh \
<workflow_parameters_file> \
<workflow_elements_file> \
<protocols_directory>
```

The command line parameters are described in Section 1. Alternatively, parameters, protocols and workflow elements can be added to the database manually in the mysql server or through the generated UI.
Run the `sh importWorksheet.sh` script with the following parameters to generate *ComputeTasks* in the database.

```
sh importWorksheet.sh \
<workflow_name> \
<worksheet_file> \
<run_id>
```

where `worksheet_file` is the worksheet file with the targets, `workflow_name` is the workflow name in the database for which you would like to generate tasks and `run_id` is the unique generation run id.

4.6 Execution on the grid with the pilot framework

We use the pilot approach to run ComputeTasks. For this, the pilot files should be present at the execution environment. In case of the grid, you need to copy three "pilot" files to the grid UI node to $HOME/maverick/ directory. These files are:

`maverick.sh` : actual pilot job, that calls back to the database and ask for available for execution ComputeTask

`maverick.jdl` : jdl file used for submission to the glite grid service (used only for the grid)

`dataTransferSRM.sh` : script to support data transfer in the grid (used only for the grid)

The files can be found at:

`https://github.com/molgenis/molgenis_apps/tree/testing/modules/compute/pilots/grid/`

The maverick.sh should be edited accordingly to your execution setting. You need to specify `back_end`, where you like to submit you ComputeTask for execution. `back_end` can have a value emphe.g. `ui.grid.sara.nl`. Also, you need to specify `your_ip` and `your_port` of your web-server, where Molgenis/compute is running.

```
   export WORKDIR=$TMPDIR
   source dataTransferSRM.sh
   curl -F status=started -F backend=back_end
your_ip:your_port/compute/api/pilot > script.sh
   bash -l script.sh 2>&1 | tee -a log.log
   curl -F status=done -F log_file=@log.log
your_ip:your_port/compute/api/pilot
```

Also, you may edit the `maverick.jdl` to specify the *walltime* and computational sites where you like to run you analysis. The example jdl requirements look like

```
   Requirements = (
   (other.GlueCEInfoHostName == "ce.lsg.psy.vu.nl" ||
   other.GlueCEInfoHostName == "ce.lsg.hubrecht.eu")
   && other.GlueCEPolicyMaxCPUTime >= 1440);
```

You Read jdl (job description language) manual for more information After putting these files in the UI node, *ComputeTasks* can be submitted with the command-line with the `sh runPilots.sh`:

```
   sh 5_runPilots.sh \ <backend> \ <username> \ <password> \
<backend_type>
```

Here, `back_end` also can value `ui.grid.sara.nl`. `username` and `password` are user grid credentials. `backend_type`, in the grid case, should have a value `grid`. It also can have a value `cluster`, that means that the PBS, SGE or BSUB scheduler is used.

There are following statuses of *ComputeTasks* in the compute database:

generated: means that the task is generated

ready: means that the task is ready for execution (all previous *ComputeTasks* are finished)

running: means that the task is running in the current moment

done means that the task is finished

After execution, the (output/error) logs of the *ComputeTasks* will be placed back in the compute database and their statuses should be *done*.

Try it out!