КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2015 Т. 7 № 3 С. 601−606

Ки&М

СЕКЦИОННЫЕ ДОКЛАДЫ

УДК: 004.75, 004.45

Running Parameter Sweep applications on Everest cloud platform

S. Yu. Volkov[a], O. V. Sukhoroslov[b]

Institute for Information Transmission Problems of the Russian Academy of Science, Kharkevich Institute, Bolshoy Karetny per. 19, build.1, Moscow 127051 Russia

E-mail: [a] fizteh.volkov@gmail.com, [b] oleg.sukhoroslov@gmail.com

Received September 30, 2014

Parameter sweep applications are a very important class of applications, which are typically defined as a set of computational experiments over a set of input parameters, each of which is executed with its own parameter combination. These computations arise in many scientific contexts. This article introduces the Parameter Sweep web service that runs such applications in distributed computing environment. Also discussed is the Everest cloud platform, on which this service is built.

Keywords: parameter sweep experiments, distributed computing, web services, cloud platform

Реализация запуска многовариантных расчетов на платформе Everest

С. Ю. Волков, О. В. Сухорослов

Институт проблем передачи информации им. А. А. Харкевича Российской академии наук, Россия, 127051, г. Москва, Большой Каретный переулок, д. 19, стр. 1

Многовариантные расчеты являются чрезвычайно важным классом приложений, обычно определяемых как набор вычислительных задач, определенных на множестве входных параметров и запускаемых с различными значениями данных параметров. Необходимость такого рода вычислений возникает во многих научных областях. Данная статья рассматривает веб-сервис, реализующий запуск данных приложений в распределенной вычислительной среде, а также облачную платформу Everest, на базе которой реализован данный сервис.

Ключевые слова: многовариантные расчеты, распределенные вычисления, веб-сервисы, облачная платформа

The work is supported by the Russian Foundation for Basic Research (grant No. 14-07-00309 A).

Citation: *Computer Research and Modeling*, 2015, vol. 7, no. 3, pp. 601–606.

Introduction

Parameter sweep applications are becoming extremely important in science and engineering. As an example, one can explore the behavior of the airfoil by running its model multiple times, depending on its properties, such as speed, angle attack, shape and so on. Parameter sweep applications address this kind of computations. They may be extremely time-consuming and require enormous amount of processor time. Therefore, this class of applications is an ideal class for distributed computing.

Parameter sweep applications involve some input set of computational parameters and files. Each parameter has its range of values, such as different angle attack values in the above example. Multiple computations, or tasks, are then run for different combinations of each parameter values. Each file may be the input of two or more tasks. Each task is supposed to have some output, typically in the form of the model's output parameters, describing the computed characteristics of this model, depending on the input parameters. The resulting set of all task outputs represents the result of the whole parameter sweep experiment.

The presented parameter sweep service has been influenced by the NimrodG system [Bethwaite et al., 2010; Buyya, Abramson and Giddy, 2000]. This system has the so-called plan file, describing the whole experiment, including parameters, input and output files and the command to be executed for each task. These tasks are generated for each combination of the parameters using the cartesian product. Our service takes the cartesian product as well, but allows users to impose some restrictions on each parameters combination in the form of the *constraints directive* (see **Plan file structure** for more details). It also allows to filter the output results and to introduce the computation's criterion to find out the 'best' (in terms of criterion) values of output parameters.

This service has been written on the Everest platform. Before we proceed to the service itself, let us introduce this cloud platform.

Everest Platform

Everest [Sukhoroslov, Afanasiev, 2014; Sukhoroslov, Rubtsov, Volkov; Everest] is a cloud platform that supports publication, sharing and reuse of scientific applications as web services. The underlying approach is based on a uniform representation of computational web services and its implementation using REST architectural style.

In contrast to traditional service development tools, Everest follows the Platform as a Service cloud delivery model by providing all its functionality via remote interfaces. A single instance of the platform can be accessed by many users in order to create, run and share services with each other without the need to install additional software on users' computers.

Another distinct feature of Everest is the ability to connect services with external computing resources. That means that service developer can provide computing resource for running service jobs. A service user can also override the default resource by providing another resource for running her jobs.

While the platform doesn't provide its own infrastructure to run compute jobs as classic PaaS examples, it can handle the problems of resource allocation, job management, data transfer and so on without the interference of users.

Everest is work in progress. The platform is currently undergoing experimental evaluation and pilot deployment.

Parameter sweep service

In order to initiate the parametric computation, the user needs to submit two files. **Figure 1** shows, how this service looks like in the client's web browser. The first one is the plan file, while the second one is the archive with the experiment's input files (currently supported are *tar.gz* and *zip* formats). If the computation succeeds, the result the user can download from the server is the archive with all of the re-

sults, satisfying both the criterion and the filter, if any. These results represent folders, containing the current task's output files as well as a parameter file, filled with the corresponding parameters values.

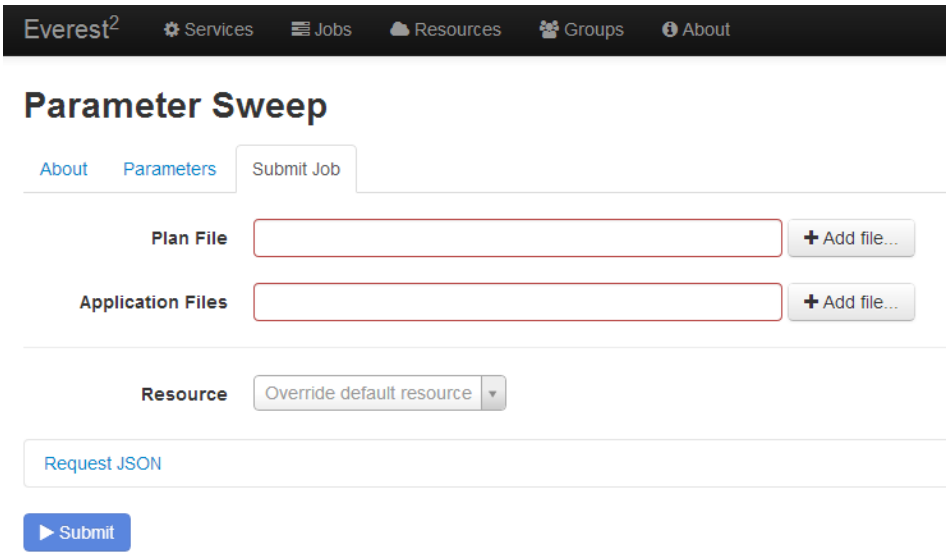

Fig. 1. Parameter Sweep web service

Let us delve a little bit into how this service works. **Figure 2** shows the architecture of the service. It's a package named *parametric*, written in the Scala language, as well as the so-called parametric application, which is implemented as a new Everest extension and has three standard methods: *prolog*, *epilog* and *taskStateChanged*. Its *prolog* method generates computational tasks (in the platform's internal format) and returns these tasks for subsequent execution by the platform on distributed computing resources. This is where the parametric extension requires plan tasks, generated from the plan file by the parametric package, and converts them to the internal format, mentioned above. The *taskStateChanged* method is invoked once some task's state has changed, for example if some task is completed or cancelled. This is a place, where the parametric application handles the task's output parameters, processing them through filters and computing the criterion's value (see **Plan file structure** for more details). Finally, the *epilog* method is invoked once all of the tasks have been completed. This is where the criterion part of the plan file, if any, is applied and the 'best' tasks are chosen and returned to the client.

Plan file structure

The syntax for our plan file has been inspired by the Nimrod's plan file. It is a simple text file, describing the experiment. It has the following directives: parameter, constraint, input, substitute, command, output, filter and criterion. The parameter, input, command and output directives are required. The constraint, substitute, filter and criterion directives are optional. Each directive, except for command, could span multiple lines. This includes either the line break, or the new line with the same directive. The directives order is significant and should be as described below.

The description of the whole computational experiment uses the standard *$var* and *${var}* substitute syntax. In our service, the curly braces '{' and '}' are optional with only one exception. If one of the substitute variables is the prefix of another, the longest one must be enclosed into curly braces. As an example, if we have two variables, *var* and *var1*, and don't enclose the second one (i.e. leave it as *$var1*), only the *$var* part will be substituted, which is obviously not what we expect.

Parameter directive. This directive describes our experiment's parameters. It has the following syntax:

parameter {name} {range}

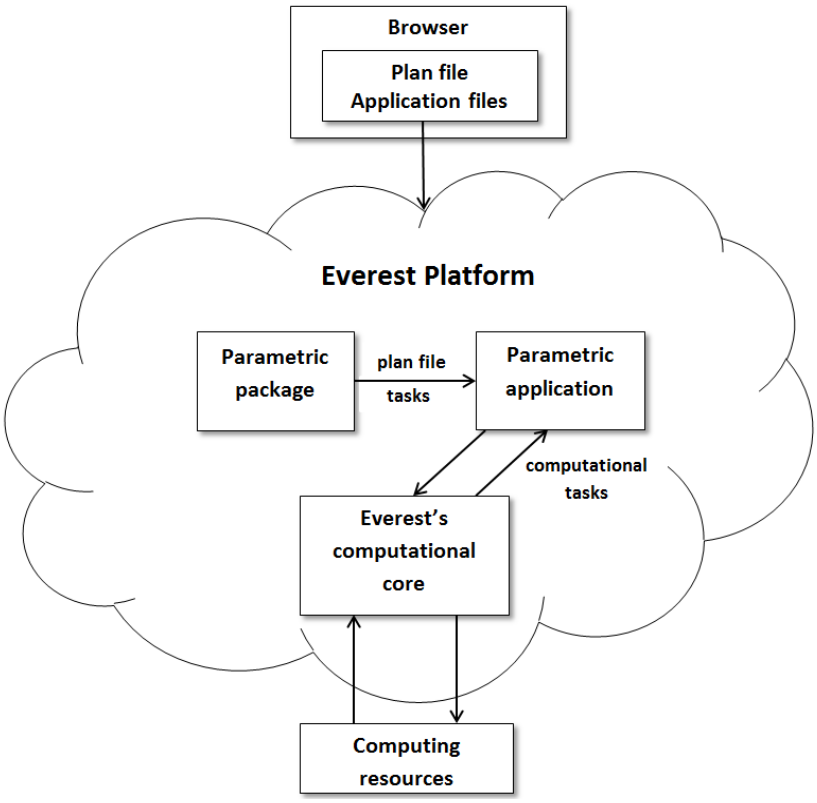

Fig. 2. Architecture of the Parameter Sweep service

The {name} is parameter's name and {range} represents the range of its values. The {range} part has the following syntax:

from {value} to {value} step {value} **or**

{list of values, separated by at least one whitespace}

The first syntax is for integers and floating point numbers only. Their values are simply integers and floating point numbers respectively. Otherwise, all of the values, separated by whitespace, should be listed. If the value itself contains at least one whitespace, it should be enclosed into quotation marks.

Constraint directive. This is an optional directive. Its goal is to impose restrictions on experiment's parameters. Its syntax is as follows:

constraint {type} {constraint expressions, separated by a comma}

The constraint expressions are math expressions. They support the following operators:

+ (addition), — (subtraction), * (multiplication), / (division), ^ (exponentiation), % (remainder operator), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), = (equal to), != (not equal to), and (conjunction), or (inclusive or), not **or** ! (negation).

They can also contain parenthesis, all standard functions like 'sin', 'cos', 'log', 'abs', 'sqrt' etc. and the names of parameters to be substituted, prefixed by the '$' sign.

The constraint's {type} specifies, whether the parameters' values or these values' indices are substituted into the constraints' expressions. It should be 'value' or 'index' respectively.

Input Files directive. This directive lists each task's input files. Here is its syntax:

input files {file names or paths, separated by at least one whitespace}

As with the **parameter** directive, these file names or paths must be enclosed into quotation marks if they contain whitespaces. Please, note that they could also be parameterized. File paths are directories inside the parameter sweep's input archive and could contain file masks using the standard * sign.

Some file names could be prefixed by the '@' sign. Such files are called *substitution files* and should contain parameters' names (specified in the **parameter** directive) with the *$* or *${}* substitution

notation. Let us assume this file is a script file, written in Scala. Here's what some part of this file may look like:

val v1 = $i
val v2 = $d
val result = someFunction(v1, v2)

Here *i* and *d* are parameters, specified in the **parameter** directive. This file will be looked through in the search of the *$var* or *${var}* syntax. If found, *$i* and *$d* will be substituted with current task's values of parameters *i* and *d*. For example, if some task has the values *i = 7* and *d = -123.32*, this part of the script will be transformed into

val v1 = 7
val v2 = -123.32
val result = someFunction(v1, v2)

It's important to notice that the parser has no idea, where this parameter could be used. If it is an *Int* or a *Boolean*, it needs to be substituted as is. However, if it is a *String*, it needs to be substituted with quotation marks. And since no one but the client himself knows, how his script works, it's up to him to take types into consideration. For example, the piece of Scala code above works if *v1* is an *Int* or a *Boolean*, but if it's a *String*, this code should be modified like val *v1 = "$i"*.

Command directive. This is the command to be executed on the computational nodes. Its syntax is the following:

command {command}

There is only one command line, which should contain only one command. Since it will be executed on the computational nodes, the Nimrod 'node: execute' declaration has been replaced with 'command'. This directive could also be parameterized.

Output Files directive. These are the task's output files. They files must be the output of the command in the previous directive, which is usually some script. As with previous directive, they could be parameterized. Here is the syntax of this directive:

output_files {file names, separated by at least one whitespace}

Some of these files could be prefixed with the '@' sign. Such files should contain the task's outputs and have the following structure:

output1 = output1Value
output2 = output2Value **and so on**.

It's important to notice that the names of the outputs must be unique. Different output files must list different outputs.

Filter directive. This directive is optional. The values of task's output parameters, described in the previous section, may be processed by the filters. Here is this directive's syntax:

filter {filter expressions, separated by a comma}

Filter expression is essentially the same as constraint expression, with only one difference: instead of parameter names to be substituted it has the outputs to be substituted. Since the names of the outputs must be unique, this directive will simply look through all of the output files until it finds the corresponding outputs. Only the tasks, satisfying all of the filter expressions, will be taken into consideration and processed by the criterion directive.

Criterion directive. This directive is optional as well. It is only applied to the tasks, satisfying the filter directive. Here is its syntax:

criterion {type} {criterion function}

The {type} part must be either *max* or *min* (case-sensitive). Criterion function is a math expression over the output parameters. The results of this directive are all of the tasks, maximizing or minimizing (depending of the {type} part) the criterion function.

Extensive example

Let us consider an example of the aforementioned directives using a well-known program of molecular docking *Autodock Vina* [Autodock Vina]. In our Parameter Sweep computation we will run 10

docking tasks with different ligands and find tasks with minimum affinity (energy). Here is the appropriate plan file:

```
parameter n from 1 to 10 step 1
input_files @run.sh vina write_score.py protein.pdbqt ligand${n}.pdbqt config.txt
command ./run.sh
output_files ligand${n}_out.pdbqt log.txt @score
criterion min $affinity
```

In this example we use **parameter** directive to define parameter *n* that refers to ligand number and takes integer values from 1 to 10.

In the **input files** directive we define input files per task. Note how *${n}* is used in the name of ligand file to refer to the value of parameter *n*. That means that task for *n=1* will use input file *ligand1.pdbqt*, the task for n=2 will use file *ligand2.pdbqt*, and so on.

In the **substitute files** directive we specify that we want to substitute all strings *$n* or *${n}* inside file *run.sh* with the value of parameter *n*. This is done on a per task level, so each task will use a different run script.

The **command** directive simply runs the script *run.sh*.

The **output files** directive lists each task's output files. In this example these are the output ligand file, the log as well as the score file, which lists the *affinity* output parameter. As already stated, this file must have the structure *affinity = affinityValue.*

Finally, we use **criterion** directive to specify that we are interested in results with minimal affinity value. The criterion function simply refers to this value as *$affinity*. This directive will look through the task's output files until it finds the *affinity* output parameter.

Conclusions and future work

This paper has covered the class of parametric applications, extremely important in many areas, such as science or engineering. We have introduced the Parameter Sweep service, built on the Everest cloud platform. Compared to prior works, it has a number of advantages, such as the possibility to filter the output results just as the client sees fit. Moreover, since it's a web service, the clients don't have to download, install and run any software. However, the work is still in progress. The main goal is to make the service more user-friendly. Among other things, this includes the web interface for building a plan file, which is definitely much more convenient than typing this file in a text editor and less error-prone. Future work will also address the problem of efficient scheduling of parameter sweep computations across multiple heterogeneous resources.

References

Autodock Vina. http://vina.scripps.edu/

Bethwaite, B., Abramson, D., Bohnert, F., Garic, S., Enticott, C., Peachey, T., "Mixing the Grid and Clouds: High-throughput Science using the Nimrod Tool Family", "Cloud Computing: Principles, Systems and Applications", Eds Antonopoulos and Gillam, Springer, pp 219-237, ISBN: 978-1-84996-240-7, 2010.

Buyya R., Abramson D. and Giddy J. "Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid", HPC Asia 2000, May 14–17, 2000. — P. 283–289, Beijing, China.

Everest. http://everest.distcomp.org/

Sukhoroslov O., Afanasiev A. Everest: A Cloud Platform for Computational Web Services // 4th International Conference on Cloud Computing and Services Science (CLOSER 2014). — P. 411–416.

Sukhoroslov O.V., Rubtsov A.O., Volkov S.Yu. Development of Distributed Computing Applications and Services with Everest Cloud Platform // In these Proceedings.