Model Exploration Using OpenMOLE - a workflow engine for large scale distributed design of experiments and parameter tuning

Romain Reuillon, Mathieu Leclaire, Jonathan Passerat-Palmbach

Model Exploration Using OpenMOLE a workflow engine for large scale distributed design of experiments and parameter tuning

Romain Reuillon [*] ,
Mathieu Leclaire[*],
Jonathan Passerat-Palmbach [†]

Originally published in: IEEE High Performance Computing and Simulation conference 2015 — July 2015 — pp n/a-n/a
to be published

[*] Institut des Systèmes Complexes Paris, Île de France
[†] Imperial College London, Department of Computing, London SW7 2AZ

RESEARCH CENTRE
Imperial College London
Department of Computing
Huxley Building
London SW7 2AZ
United Kingdom

Abstract: OpenMOLE is a scientific workflow engine with a strong emphasis on workload distribution. Workflows are designed using a high level Domain Specific Language (DSL) built on top of Scala. It exposes natural parallelism constructs to easily delegate the workload resulting from a workflow to a wide range of distributed computing environments. In this work, we briefly expose the strong assets of OpenMOLE and demonstrate its efficiency at exploring the parameter set of an agent simulation model. We perform a multi-objective optimisation on this model using computationally expensive Genetic Algorithms (GA). OpenMOLE hides the complexity of designing such an experiment thanks to its DSL, and transparently distributes the optimisation process. The example shows how an initialisation of the GA with a population of 200,000 individuals can be evaluated in one hour on the European Grid Infrastructure.

Keywords: Distributed computing; Scientific workflows; Model Exploration; Design of Experiments; Evolutionary Algorithms

1 Introduction

Parameter tuning is a daily problem in any scientific community using complex algorithms. In the specific case of simulation applications, increased performance of computer architectures have led to more and more ambitious models with a growing number of parameters. Therefore exploring high dimensional spaces to tune parameters for specific problems has become a central problem. Stochastic models add another dimension of parameters to explore, as different random sources should generally be tested for each set of parameters, in order to obtain statistically sound results.

This wide range of parameters to tune, combined with the intrinsic execution time of the application, make it impossible to run significant Designs of Experiments (DoE) on a desktop computer. Experiments within a DoE can be processed independently from each other. They are perfect candidates for distributed computing platforms. In an ideal world, the time required to process the whole DoE would be almost equivalent to the execution time of a single experiment. However the methodological and technical costs of using distributed execution environments imply that most parameter space explorations are achieved either on a single desktop computer and occasionally on a multi-core server with shared memory. Larger scale platforms are rarely used, although clusters or worldwide computing infrastructures like EGI (European Grid Initiative) are well suited for this kind of applications.

Compared to other workflow processing engines, OpenMOLE promotes a zero-deployment approach by accessing the computing environments from bare metal, and copies on-the-fly any software component required for a reliable remote execution. OpenMOLE also encourages the use of software components developed in heterogeneous programming languages and enables users to easily replace the elements involved in the workflow. Workflows can be designed using either a Graphical User Interface (GUI), or a Domain Specific Language (DSL) which exposes advanced workflow design constructs.

Apart from these core elements of the platform, OpenMOLE ships with its own software ecosystem. It contains among others GridScale[a], a library to access a wide range of computing environment, and Yapa[b], a packaging tool ensuring the successful re-execution of applications across heterogeneous platforms. For more details regarding the core implementation and features of OpenMOLE, interested readers can refer to [Reuillon et al., 2010, Reuillon et al., 2013] and the OpenMOLE website[Reuillon et al., 2015].

OpenMOLE focuses on making distributed computing available in the most straightforward way to the scientific community. Depending on the applications, several problems might arise when attempting to distribute the execution. Software tools can help scientists overcome these barriers. This paper describes the input of OpenMOLE and its software ecosystem to the distribution of complex scientific applications to remote execution environments.

[a] `https://github.com/openmole/gridscale`
[b] `https://github.com/openmole/yapa`

We first detail the problems faced by the scientific community to effectively distribute an application. Then, we present how the tools from the OpenMOLE ecosystem can answer these problems. The last section presents a test case showing how an actual simulation application was successfully distributed using OpenMOLE.

2 What is OpenMOLE?

OpenMOLE is a scientific workflow engine with facilities to delegate its workload to a wide range of distributed computing environments. It shows several main advantages with respect to the other workflow management tools available. First, OpenMOLE distinguishes as a tool that does not target a specific scientific community, but offers generic tools to explore large parameter sets.

Second, OpenMOLE features a Domain Specific Language (DSL) to describe the workflows. According to [Barker and Van Hemert, 2008], workflow platforms should not introduce new languages but rely on established ones. OpenMOLE's DSL is based on the high level Scala programming language [Odersky et al., 2004]. In addition to the DSL, a web interface is currently under development, and will permit expressing workflows graphically.

Finally, OpenMOLE features a great range of platforms to distribute the execution of workflows, thanks to the underlying GridScale library[c]. GridScale is part of the OpenMOLE ecosystem and acts as one of its foundation layers. It is responsible for accessing the different execution environments. The last release of OpenMOLE can target SSH servers, multiple cluster managers and computing grids ruled by the gLite/EMI middleware.

In this section, we describe two main components of the OpenMOLE platform: the Domain Specific Language and the distributed environments. They contribute to make the exploration of a parameter set simple to distribute.

2.1 A DSL to describe workflows

Scientific experiments are characterised by their ability to be reproduced. This implies capturing all the processing stages leading to the result. Many execution platforms introduce the notion of workflow to do so [Barker and Van Hemert, 2008, Mikut et al., 2013]. Likewise, OpenMOLE manipulates workflows and distribute their execution across various computing environments.

A workflow is a set of tasks linked with each other through transitions. From a high level point of view, tasks comprise inputs, outputs and optional default values. Tasks describe what OpenMOLE should execute and delegate to remote environments. They embed the actual applications to study. Depending on the kind of program (binary executable, Java...) to embed in OpenMOLE, the user chooses the corresponding task. Tasks execution depends on inputs variables, which are provided by the dataflow. Each task produces outputs returned to the dataflow and transmitted to the input of consecutive tasks. OpenMOLE

[c] `https://github.com/openmole/gridscale`

exposes several facilities to inject data in the dataflow (*sources*) and extract useful results at the end of the experiment (*hooks*).

Two choices are available when it comes to describe a workflow in OpenMOLE: the Graphical User Interface (GUI) and the Domain Specific Language (DSL). Both strategies result in identical workflows. They can be shared by users as a way to reproduce their execution.

OpenMOLE's DSL is based upon the Scala programming language, and embeds new operators to manage the construction and execution of the workflow. The advantage of this approach lies in the fact that workflows can exist even outside the OpenMOLE environment. As a high-level language, the DSL can be assimilated to an algorithm described in pseudocode, understandable by most scientists. Moreover, it denotes all the types and data used within the workflow, as well as their origin. This reinforces the capacity to reproduce workflow execution both within the OpenMOLE platform or using another tool.

The philosophy of OpenMOLE is *test small* (on your computer) and *scale for free* (on remote distributed computing environments). The DSL supports all the Scala constructs and provides additional operators and classes especially designed to compose workflows. OpenMOLE workflows expose explicit parallel aspects of the workload that can be delegated to distributed computing environments in a transparent manner. The next sections introduces the available computing environments.

2.2 Distributed Computing environments

OpenMOLE helps delegate the workload to a wide range of HPC environments including remote servers (through SSH), clusters (supporting the job schedulers PBS, SGE, Slurm, OAR and Condor) and computing grids running the gLite/EMI middleware.

Submitting jobs to distributed computing environments can be complex for some users. This difficulty is hidden by the GridScale library from the OpenMOLE ecosystem. GridScale provides a high level abstraction to all the execution platforms mentioned previously.

When GridScale was originally conceived, a choice was made not to rely on a standard API (Application Programming Interface) to interface with the computing environments, but to take advantage of the command line tools available instead. As a result, GridScale can embed any job submission environment available from a command line. From a higher perspective, this allows OpenMOLE to work seamlessly with any computing environment the user can access.

Users are only expected to select the execution environment for the tasks of the workflow. This choice can be guided by two considerations: the availability of the resources and their suitability to process a particular problem. The characteristics of each available environment must be considered and matched with the application's characteristics. Depending on the size of the input and output data, the execution time of a single instance and the number of independent executions to process, some environments might show more appropriate than others.

At this stage, OpenMOLE's simple workflow description is quite convenient to determine the computing environment best suited for a workflow. Switching from one environment to another is achieved either by a single click (if the workflow was designed with the GUI) or by modifying a single line (for workflows described using the DSL).

Some applications might show more complicated than others to distribute. The next section exposes the main challenges a user is faced with when trying to distribute an application. We present how OpenMOLE couples with a third-party software called CARE to solve these problems.

3 The Challenges of Distributing Applications

3.1 Problems and classical solutions

Let us consider all the dependencies introduced by software bundles explicitly used by the developer. They can take various forms depending on the underlying technology. Compiled binary applications will rely on shared libraries, while interpreted languages such as Python will call other scripts stored in packages.

These software dependencies become a problem when distributing an application. It is indeed very unlikely that a large number of remote hosts are deployed in the same configuration as a researcher's desktop computer. Actually, the larger the pool of distributed machines, the more heterogeneous they are likely to be.

If a dependency is missing at runtime, the remote execution will simply fail on the remote hosts where the requested dependencies are not installed. An application can also be prevented from running properly due to incompatibilities between versions of the deployed dependencies. This case can lead to silent errors, where a software dependency would be present in a different configuration and would generate different results for the studied application.

Silent errors break Provenance, a major concern of the scientific community [Miles et al., 2007, MacKenzie-Graham et al., 2008]. Provenance criteria are satisfied when an application is documented thoroughly enough to be reproducible. This can only happen in distributed computing environments if the software dependencies are clearly described and available.

Some programming environments provide a solution to these problems. Compiled languages such as C and C++ offer to build a static binary, which packages all the software dependencies. Some applications can be very difficult to compile statically. A typical case is an application using a closed source library, for which only a shared library is available.

Another approach is to rely on an archiving format specific to a programming language. The most evident example falling into this category are Java Archives (JAR) that embed all the Java libraries an application will need.

A new trend coming from recent advances in the software engineering community is embodied by Docker. Docker has become popular with DevOps techniques to improve software developers efficiency. It enables them to ship their

application within a so-called container that will include the application and its required set of dependencies. Containers can be transferred just like an archive and re-executed on another Docker engine. Docker containers run in a sandboxed virtual environment but they are not to be confound with virtual machines. They are more lightweight as they don't embed a full operating system stack. The use of Docker for reproducible research has been tackled in [Chamberlain et al., 2014].

The main drawback of Docker is that it implies deploying a Docker engine on the target host. Having a Docker engine running on every target host is a dodgy assumption in heterogeneous distributed environments such as computing grids.

The last option is to rely on a third-party application to generate re-executable applications. The strategy consists in collecting all the dependencies during a first execution in order to store them in an archive. This newly generated bundle is then shipped to remote hosts instead of the original application. This is the approach championed by tools like CDE [Guo, 2012] or CARE [Janin et al., 2014].

Considering all these aspects, the OpenMOLE platform has for long chosen to couple with tools providing standalone packages. While CDE was the initial choice, recent requirements in the OpenMOLE user community have led the development team to switch to the more flexible CARE. The next section will detail how OpenMOLE relies on CARE to package applications.

3.2 Combining OpenMOLE with CARE

The first step towards spreading the workload across heterogeneous computing elements is to make the studied application executable on the greatest number of environments. We have seen previously that this could be difficult with the entanglement of complex software environments available nowadays. For instance, a Python script will run only in a particular version of the interpreter and may also make use of binary dependencies. The best solution to make sure the execution will run as seamlessly on a remote host as it does on the desktop machine of the scientist is to track all the dependencies of the application and ship them with it on the execution site.

OpenMOLE used to provide this feature through a third-party tool called CDE (Code, Data, and Environment packaging) [Guo, 2012]. CDE creates archives containing all the items required by an application to run on any recent Linux platform. CDE tracks all the files that interact with the application and creates the base archive.

The only constraint regarding CDE is to create the archive on a platform running a Linux kernel from the same generation as those of the targeted computing elements. As a rule of thumb, a good way to ensure that the deployment will be successful is to create the CDE package from a system running Linux 2.6.32. Many HPC environments run this version, as it is the default kernel used by science-oriented Linux distribution, such as Scientific Linux and CentOS.

CARE on the other hand presents more advanced features than CDE. CDE actually displays the same limit than a traditional binary run on a remote host:

i.e. the archive has to be generated on a platform running an old enough Linux kernel, to have a maximum compatibility with remote hosts. CARE lifts this constraint by emulating missing system calls on the remote environment. Thus, an application packaged on a recent release of the Linux kernel will successfully re-execute on an older kernel thanks to this emulation feature. Last but not least, CDE's development has been stalled over the last few years, whereas CARE was still actively developed over the last few months. CARE's developers are also very reactive when an eventual problem is detected in their piece of software. This makes CARE a reliable long-term choice for re-execution facilities. All that remains is to complete the package by adding specific customisations related to the integration of the application within an OpenMOLE workflow.

As previously evoked, OpenMOLE workflows are mainly composed of tasks. Different types of tasks exist, each embedding a different kind of application. Generic applications such as those packaged with CARE are handled by the *SystemExecTask*. As an OpenMOLE task, the generated element is ready to be added to the OpenMOLE scene and integrated in a workflow.

We will now demonstrate the use of the DSL and computing environments with a concrete example. For the sake of simplicity, we will exploit a simulation model developed with the NetLogo[Wilensky, 1999] platform. This simulation platform benefits from a native integration in OpenMOLE, which ensures the model will run on remote hosts. It spares the user from the extra packaging step using CARE that was introduced in this section.

4 An A to Z example: Calibrating a model using Genetic Algorithms

This example presents step by step how to explore a NetLogo model with an Evolutionary/Genetic Algorithm (EA/GA) in OpenMOLE. We've chosen NetLogo for its simplicity to design simple simulation models with a graphical output quickly. However, this approach can be applied to any other kind of simulation model, regardless of their implementation platform.

4.1 The ant model

We demonstrate this example using the ants foraging model present in the Netlogo library. This model was created by Ury Wilensky. According to NetLogo's website, this model is described as:
"*In this project, a colony of ants forages for food. Though each ant follows a set of simple rules, the colony as a whole acts in a sophisticated way. When an ant finds a piece of food, it carries the food back to the nest, dropping a chemical as it moves. When other ants "sniff" the chemical, they follow the chemical toward the food. As more ants carry food to the nest, they reinforce the chemical trail."*

A visual representation of this model appears in Figure 1.

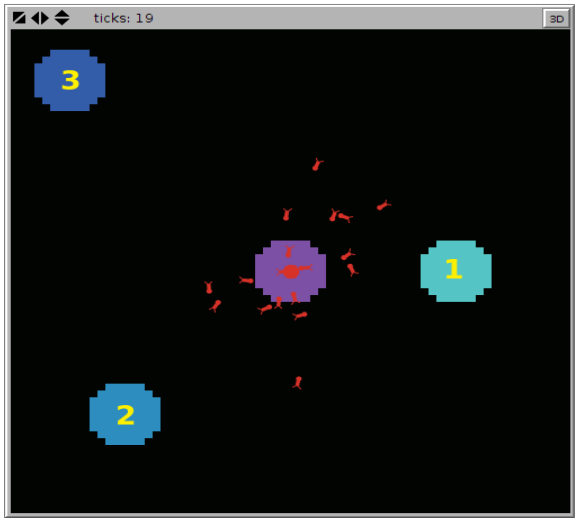

Figure 1: **Visual Representation of the Ant Model Showing the 3 Food Sources and the Multiple Ant Agents**

In this tutorial we use a headless version of the model. This modified version is available from the OpenMOLE's website[d].

[d] `http://www.openmole.org/current/ants.nlogo`

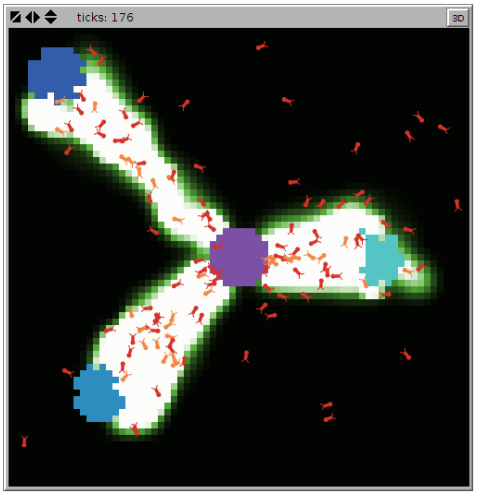

Figure 2: **Graphical Output of the Ants Agents Foraging from 3 Different Sources**

4.2 Define the problem to solve as an optimisation problem

This model manipulates three parameters:

- *Population*: number of ants in the model,
- *Evaporation-rate*: controls the evaporation rate of the chemical,
- *Diffusion-rate*: controls the diffusion rate of the chemical.

Ants forage from three sources of food as represented in Figure 2). Each source is positioned at different distances from the ant colony.

In this example, we want to search the best combination of the two parameters *evaporation-rate* and *diffusion-rate* which minimises the eating time of each food source. We will use OpenMOLE's embedded Evolutionary Algorithms features to perform this optimisation process. The first thing is to define a fitness function describing the optimisation problem.

We build our fitness function by modifying the NetLogo Ants source code to store for each food source the first ticks indicating that this food source is empty, as shown in Listing 1:

```
to compute-fitness
  if ((sum [food] of patches with [food-source-number = 1] = 0)
      and (final-ticks-food1 = 0)) [
    set final-ticks-food1 ticks ]
```

```
  if ((sum [food] of patches with [food-source-number = 2] = 0)
      and (final-ticks-food2 = 0)) [
    set final-ticks-food2 ticks ]
  if ((sum [food] of patches with [food-source-number = 3] = 0)
      and (final-ticks-food3 = 0)) [
    set final-ticks-food3 ticks ]
end
```

Listing 1: **NetLogo Function Returning the Simulation Tick at Which each Food Source Became Empty**

At the end of each simulation we return the values for the three objectives:

- The simulation ticks indicating that source 1 is empty,
- The simulation ticks indicating that source 2 is empty,
- The simulation ticks indicating that source 3 is empty.

The combination of the three objectives indicates the quality of the parameters used to run the simulation. This situation is a multi-objective optimisation problem. In case there is a compromise between these three objectives, we will obtain a Pareto frontier at the end of the optimisation process.

4.3 Getting the ant model to run in OpenMOLE

When building a calibration or optimisation workflow, the first step is to make the model run in OpenMOLE. The script displayed in Listing 2 simply embeds the NetLogo model and runs one single execution of the model with arbitrary parameters.

```
// Define the input variables
val gPopulation = Val[Double]
val gDiffusionRate = Val[Double]
val gEvaporationRate = Val[Double]
val seed = Val[Int]

// Define the output variables
val food1 = Val[Double]
val food2 = Val[Double]
val food3 = Val[Double]

// Define the NetlogoTask
val cmds = Seq("random-seed ${seed}", "run-to-grid")
val ants =
  NetLogo5Task("Ants.nlogo", cmds) set (
    // Map the OpenMOLE variables to NetLogo variables
    netLogoInputs += (gPopulation, "gpopulation"),
```

```
    netLogoInputs += (gDiffusionRate, "gdiffusion-rate"),
    netLogoInputs += (gEvaporationRate, "gevaporation-rate"),
    netLogoOutputs += ("final-ticks-food1", food1),
    netLogoOutputs += ("final-ticks-food2", food2),
    netLogoOutputs += ("final-ticks-food3", food3),
    // The seed is used to control the initialisation of the random
        number generator of NetLogo
    inputs += seed,
    // Define default values for inputs of the model
    seed := 42,
    gPopulation := 125.0,
    gDiffusionRate := 50.0,
    gEvaporationRate := 50
  )

// Define the hooks to collect the results
val displayHook = ToStringHook(food1, food2, food3)

// Start a workflow with 1 task
val ex = (ants hook displayHook) start
```

Listing 2: **Complete OpenMOLE Workflow Embedding the Ant Model**

The code snippet in Listing 2 introduces several notions from OpenMOLE. First, the original model is wrapped in a *NetLogoTask*. It is of course not the case for all the different simulation frameworks. The two other main types of tasks are the *ScalaTask*, that executes inline Scala code, and the *SystemExecTask*, which runs any kind of application as it would be from a command line.

The second notion to observe is the Hook *displayHook* associated with the main task. Tasks are mute pieces of software. They are not conceived to write files, display values, nor more generally present any side effects at all. The role of tasks is to compute some output data from their input data. That's what guaranties that their execution can be delegated to other machines.

OpenMOLE introduces a mechanism called *Hooks* to save or display results generated on remote environments. Hooks are conceived to perform an action upon completion of the task they are attached to. In this example, we use a *ToStringHook* that displays the value of the task's outputs.

4.4 Managing the stochasticity

Generally agents models, such as the one we're studying, are stochastic. It means that their execution depends on the realisation of random variates. This makes their output variables random variates as well. These random variates can be studied by estimating their distribution.

Getting one single realisation of the output random variates doesn't provide enough information to estimate their distribution. As a consequence, the model

must be executed several times, with different random sources. All these executions should be statistically independent to ensure the independent realisation of the model's output random variates. This operation is called "replications".

OpenMOLE provides the necessary mechanisms to easily replicate executions and aggregate the results using a simple statistical descriptor. The script in Listing 3 executes the ants model five times, and computes the median of each output. The median is a statistical descriptor of the outputs of the model (however, the form of the distribution remains unknown).

Replicating a stochastic experiment only five times is generally unreliable. Five is here an arbitrary choice to reduce the global execution time of this toy example.

```
val modelCapsule = Capsule(ants)

// Define the output variables
val medNumberFood1 = Val[Double]
val medNumberFood2 = Val[Double]
val medNumberFood3 = Val[Double]

// Compute three medians
val statistic =
  StatisticTask() set (
    statistics += (food1, medNumberFood1, median),
    statistics += (food2, medNumberFood2, median),
    statistics += (food3, medNumberFood3, median)
  )

val statisticCapsule = Capsule(statistic)

val seedFactor = seed in (UniformDistribution[Int]() take 5)
val replicateModel = Replicate(modelCapsule, seedFactor,
    statisticCapsule)

// Define the hooks to collect the results
val displayOutputs = ToStringHook(food1, food2, food3)
val displayMedians = ToStringHook(medNumberFood1, medNumberFood2
    , medNumberFood3)

// Execute the workflow
val ex = replicateModel + (modelCapsule hook displayOutputs) + (
    statisticCapsule hook displayMedians) start
```

Listing 3: **Median Computation on the Ant Model in OpenMOLE**

4.5 The optimisation algorithm

Now that we have estimators of the output distribution, we will try to find the parameter settings minimising these estimators. Listing 4 describes how to use the NSGA2 multi-objective optimisation algorithm [Deb et al., 2002] in OpenMOLE. The result files are written to */tmp/ants*.

```
// Define the population (10) and the number of generations (100).
// Define the inputs and their respective variation bounds.
// Define the objectives to minimize.
// Assign 1 percent of the computing time to reevaluating
// parameter settings to eliminate over-evaluated individuals.
val evolution =
  NSGA2(
    mu = 10,
    termination = 100,
    inputs = Seq(gDiffusionRate -> (0.0, 99.0), gEvaporationRate
        -> (0.0, 99.0)),
    objectives = Seq(medNumberFood1, medNumberFood2,
        medNumberFood3),
    reevaluate = 0.01
  )

// Define a builder to use NSGA2 generational EA algorithm.
// replicateModel is the fitness function to optimise.
// lambda is the size of the offspring (and the parallelism level).
val nsga2 =
  GenerationalGA(evolution)(
    replicateModel,
    lambda = 10
  )

// Define a hook to save the Pareto frontier
val savePopulationHook = SavePopulationHook(nsga2, "/tmp/ants/")

// Define another hook to display the generation in the console
val display = DisplayHook("Generation ${" + nsga2.generation.
    name + "}")

// Plug everything together to create the workflow
val ex = nsga2.puzzle + (nsga2.output hook savePopulationHook
    hook display) start
```

Listing 4: **Parameter Optimisation Using the NSGA-II Genetic Algorithm in OpenMOLE**

4.6 Scale up

When the necessity comes to scale up and expand the exploration, OpenMOLE's environments come very handy to quickly distribute the workload of the workflow to a large computing environment such as the European Grid Infrastructure (EGI). The optimisation as we've done so far is not perfectly suited for this kind of remote environments. In this case, we'll use the Island model.

Islands are better suited to exploit distributed computing resources than classical generational genetic algorithms. Islands of population evolve for a while on a remote node. When an island is finished, its final population is merged back into a global archive. A new island is then generated until the termination criterion is met: i.e. the total number of islands to generate has been reached.

Listing 5 shows that implementing islands in the workflow leaves the script almost unchanged, save for the island and environment definition. Here we compute 2,000 islands in parallel, each running for 1 hour on the European grid:

```
// Define the population (200) and the computation time (1h)
// The remaining is the same as above
val evolution =
  NSGA2(
    mu = 200,
    termination = Timed(1 hour),
    inputs = Seq(gDiffusionRate -> (0.0, 99.0), gEvaporationRate
        -> (0.0, 99.0)),
    objectives = Seq(medNumberFood1, medNumberFood2,
        medNumberFood3),
    reevaluate = 0.01
  )

// Define the island model with 2,000 concurrent islands. Each island
    gets 50 individuals sampled from the global
// population. The algorithm stops after 200,000 islands evaluations.
val (ga, island) = IslandSteadyGA(evolution, replicateModel)
    (2000, 200000, 50)

val savePopulationHook = SavePopulationHook(ga, "/tmp/ants/")
val display = DisplayHook("Generation ${" + ga.generation.name
    + "}")

// Define the execution environment
val env = EGIEnvironment("biomed", openMOLEMemory = 1200,
    wallTime = 4 hours)

// Define the execution
```

```
val ex =
  (ga.puzzle +
   (island on env) +
(ga.output hook savePopulationHook hook display)) start
```

Listing 5: **Distribution of the Parameter Optimisation Process Using the Islands Model**

5 Conclusion

In this paper, we have shown the features and capabilities of the OpenMOLE scientific workflow engine.

The light was put on two main components of OpenMOLE: its Domain Specific Language and the set of distributed environments it can address. The DSL is an elegant and simple way to describe scientific workflows from any field of study. The described workflows can then be executed on a wide range of distributed computing environments including the most popular job schedulers and grid middlewares.

The DSL and computing environment were then applied to a real-life Ant simulation model. We showed how to describe a multi-objective optimisation problem in OpenMOLE, in order to optimise a particular parameter from the model. The resulting workload was delegated to the European Grid Infrastructure (EGI).

OpenMOLE as well as all the tools forming its ecosystem are free and open source software. This allows anyone to contribute to the main project, or build extensions on top of it.

Future releases of the OpenMOLE platform will integrate a fully functional web user interface to design workflows, with the DSL still playing a key part in the design.

Acknowledgment

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 319456.

References

[Barker and Van Hemert, 2008] Barker, A. and Van Hemert, J. (2008). Scientific workflow: a survey and research directions. In *Parallel Processing and Applied Mathematics*, pages 746–753. Springer.

[Chamberlain et al., 2014] Chamberlain, R., Invenshure, L., and Schommer, J. (2014). Using Docker to support reproducible research. Technical report,

Technical Report 1101910, figshare, 2014. http://dx. doi. org/10.6084/m9. figshare. 1101910.

[Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197.

[Guo, 2012] Guo, P. (2012). CDE: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35.

[Janin et al., 2014] Janin, Y., Vincent, C., and Duraffort, R. (2014). CARE, the comprehensive archiver for reproducible execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, page 1. ACM.

[MacKenzie-Graham et al., 2008] MacKenzie-Graham, A. J., Van Horn, J. D., Woods, R. P., Crawford, K. L., and Toga, A. W. (2008). Provenance in neuroimaging. *NeuroImage*, 42(1):178–195.

[Mikut et al., 2013] Mikut, R., Dickmeis, T., Driever, W., Geurts, P., Hamprecht, F. A., Kausler, B. X., Ledesma-Carbayo, M. J., Marée, R., Mikula, K., Pantazis, P., Ronneberger, O., Santos, A., Stotzka, Rainer, Strähle, Uwe, and Peyriéras, Nadine (2013). Automated Processing of Zebrafish Imaging Data: A Survey. *Zebrafish*, 10(3):401–421.

[Miles et al., 2007] Miles, S., Groth, P., Branco, M., and Moreau, L. (2007). The requirements of using provenance in e-science experiments. *Journal of Grid Computing*, 5(1):1–25.

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the Scala programming language. Technical report, Technical Report IC/2004/64, EPFL Lausanne, Switzerland.

[Reuillon et al., 2010] Reuillon, R., Chuffart, F., Leclaire, M., Faure, T., Dumoulin, N., and Hill, D. (2010). Declarative task delegation in OpenMOLE. In *High performance computing and simulation (hpcs), 2010 international conference on*, pages 55–62. IEEE.

[Reuillon et al., 2015] Reuillon, R., Leclaire, M., and Passerat-Palmbach, J. (2015). OpenMOLE website.

[Reuillon et al., 2013] Reuillon, R., Leclaire, M., and Rey-Coyrehourcq, S. (2013). OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems*, 29(8):1981–1990.

[Wilensky, 1999] Wilensky, U. (1999). NetLogo (and NetLogo User Manual).

RESEARCH CENTRE
Imperial College London
Department of Computing
Huxley Building
London SW7 2AZ
United Kingdom

Publisher
Imperial College London
Department of Computing
Huxley Building
London SW7 2AZ
United Kingdom
http://www3.imperial.ac.uk/