Towards a high level programming paradigm to deploy e-science applications with dynamic workflows on large scale distributed systems

Mohamed Ben Belgacem
University of Geneva
CUI, Route de Drize 7,
CH-1227, Carouge, Switzerland
Email: mohamed.benbelgacem@unige.ch

Nabil Abdennadher
University of Applied Sciences, Western Switzerland, hepia
Rue de la Prairie 4,
CH-1202 Geneva, Switzerland
Email: nabil.abdennadher@hesge.ch

***Abstract*—This papers targeted scientists and programmers who need to easily develop and run e-science applications on large scale distributed systems. We present a rich programming paradigm and environment used to develop and deploy high performance applications (HPC) on large scale distributed and heterogeneous platforms. We particularly target iterative e-science applications where (i) convergence conditions and number of jobs are not known in advance, (ii) jobs are created on the fly and (iii) jobs could be persistent.**

We propose two programming paradigms so as to provide intuitive statements enabling an easy writing of HPC e-science applications. Non-expert developers (scientific researchers) can use them to guarantee fast development and efficient deployment of their applications.

I. Introduction

High Performance Computing (HPC) landscape has radically changed since two decades. Different hardwares, ranging from massively parallel to large scale distributed platforms, have been proposed. This has led to a gap increase between existing programming methodologies for scientific applications and the jungle of available and heterogeneous computing resources. Besides, the development of parallel and distributed applications is a difficult and challenging task for two reasons. Firstly, scientists have to deal with the heterogeneity of available resources. Secondly, writing new applications and/or porting legacy applications on new parallel/distributed platforms requires handling non-trivial programming difficulties, such as synchronization, load-balancing, distributed submission of tasks, etc. Non IT scientists are increasingly "forced" to master high programming skills instead of focusing on their science.

Today, there is still no programming paradigm for large scale distributed and heterogeneous infrastructures that cover all families of parallel/distributed applications: master/slave, task farming, tightly coupled applications, embarrassingly parallel applications, static and/or dynamic workflow applications, etc. Some programming paradigms support some of these families and target a limited number of distributed platforms, but none of them supports all of these families.

In addition, to our best knowledge, no programming paradigm exists to easily develop and execute e-science applications with unknown "in advance convergence condition" on large scale distributed systems. In what follows, we call these applications "Dynamic Workflow (DW) e-science applications".

DW applications are used in various domains such as phylogeny, genetics, biotechnology, medicine and environment.

This paper proposes two intuitive and easy programming paradigms to develop dynamic workflow (DW) e-science applications and deploy them on large scale distributed and heterogeneous infrastructures. The two proposed models enrich an existing toolkit prototype [1] that supports two families of parallel and distributed applications: master/slave and task farming.

The DW e-science applications we are targeting have the following features:

1) An iterative model based on a master/slave architecture.
2) The number of iterations and slave jobs per iteration are not known in advance.
3) The master job submits slave jobs, provides them with input data, retrieves their results and checks if the convergence condition is reached.
4) Slave jobs receive input data from master jobs, process them and return back the results.
5) Slave jobs could be persistent or not. Persistent jobs remain running after sending their results to the master job. They continue generating new results even if they do not receive new input data. Communications between two persistent jobs can occur in order to exchange boundary data, whereas non persistent jobs stop running after sending their results. At this stage, it's worth noting that persistency, in our context, is not an implementation detail. It is tightly related to the "business" of the application.

Unlike MapReduce, the two models proposed in this paper have the following characteristics:

1) The number of slave jobs are not known in advance and can vary during execution. In MapReduce, the number of jobs is fixed in advance.
2) The convergence condition is not known in advance, while in MapReduce there is no convergence condition.

3) Slave jobs can remain running (persistent jobs) even after sending their results. In MapReduce, the slave jobs stop running as soon as they deliver their output.

The paper is organized as follows:

Section II details some prototypes and research projects aiming at designing programming paradigms for parallel and distributed applications.

Section III presents two paradigm models used to develop and deploy DW e-science applications on large scale distributed systems.

In section IV, both models are used to implement and deploy two examples of DW e-science applications (MetaPIGA and NeuroWeb) coming from two different domains : phylogeny and medicine. The section gives emphasis to the intuitive aspect and simplicity of programming offered by our model when developing these applications.

Section V compares our model with another existing paradigm : GC3Pie [2], [3] in term of overhead and easiness. The last section draws the concluding remarks and perspectives.

II. Related works

Currently several parallel programming models and paradigms exist to port applications on HPC infrastructures. Message Passing Interface (MPI) model is widely used in scientific and engineering domains. However, this paradigm requires high skills in parallel programing where the parallelization and communication operations are explicitly handled and coded by the programmer.

Another interesting alternative to port applications on distributed parallel infrastructures is the "programming skeleton algorithm". It is based on the well-known parallel template paradigms. This high-level parallel programming technique enables to develop programs by composing a set of skeletons. The programmer adheres to top-down design and construction where the control is inherited through the structure [4]. The skeleton proposes a set of interfaces to end-users allowing them to design their computational parallel applications. These interfaces hide the implementation details and can be implemented on top of MPI, shared memory, object oriented programming or any other parallel programming model. There exist an algorithmic skeleton libraries implementation that aims at writing efficient parallel programs, for instance, Skandiuam [5] and Cilk Plus [6].

Many works have been done to give scientists easy access to distributed parallel computation. The work in [7] proposed a parallel implementation of the shallow water equation using a skeleton library called "SkelGIS". Authors showed that this library has the same performance as an MPI implementation. In addition, "SkelGIS" is more efficient and easier in term of coding and learning efforts. In [8], authors compared the programming models for data intensive systems on large computing clusters, such as Hadoop [9], distributed SQL database, and DryadLINQ [10]. They showed that the choice of programming interfaces has an effect on the code readability and computation performance. In [11], the authors proposed a portable high level programming interface for programming complex scientific applications on Graphics Processing Unit (GPU). They introduced the "SkelCL" library that provides four basic programming skeletons: *Map*, *Zip*, *Reduce* and *Scan* that implicitly handle the synchronization and data transfer. Even though "SkelCL" adds a minor performance overhead of less than 5% compared to CUDA [12] and OpenCL [13] libraries, it significantly reduces the programming effort. In the domain of integrated circuit design, [14] proposes a skeleton based library as a solution for reconfiguring FPGA devices. The reconfiguring FPGA is similar to applying different modules in parallel on different electronic regions. Each region is seen as a machine node with reference to a computing cluster. The skeleton is used as a bridge between circuit design and the applications development for FPGA.

GC3Pie [2], [3] is a High-Throughput framework to run parallel and large distributed application over HPC systems. It is a library of Python classes used to remotely run and monitor jobs on diverse batch executing systems (e.g., *Torque* [15], *Slurm* [16]), *ARC* grid midlleware [17] and cloud platforms (e.g., OpenStack [18] and Amazon-EC2). The main objective of GC3Pie is to relieve the complexity of executing and monitoring of jobs on HPC resources and move most decision making logic to the application level. Using GC3Pie, one can easily write a Python program that generates a dynamic computational workflow through using loops and conditionally branch execution for example.

In [19], [20], the authors proposed an API (XWCH API) used to develop e-science applications. XWCH API is built on top of several large scale distributed and heterogeneous infrastructures. It has been used to develop and port several e-science applications : Phylip [21], NeuroWeb [20], GIFT [22], Cyclone [23], MetaPIGA [24], [25], Selector [26] and SWAT [27]. Some of these packages belong to the DW applications family.

The use of XWCH API by non-expert computer developers is not trivial: source codes are often too verbose and difficult to understand and maintain.

In order to provide scientists with an easy development toolkit for e-science applications, two simple paradigms which target task-farming and master/slave applications are proposed [1]. These paradigms, built on top of the XWCH API are considered as a high level API hiding the underlying complexity. Even though they are simple to use, these two models do not support DW e-science applications.

Below, we propose two simple and intuitive paradigms to write DW e-science applications. The proposed paradigms hide the complexity of parallel and distributed programming and are very close to the "business logic" of scientists.

III. New paradigms for DW e-science applications

We consider a computational dynamic workflow as defined in section I: number of iterations and number of jobs per iteration are not known in advance and the convergence condition relies on the in-time results of jobs. For a given application, jobs can be persistent or non persistent.

We propose two patterns (distributed programming paradigms) depicted in Fig. 1 which target DW applications with (i) non persistent and (ii) persistent jobs.

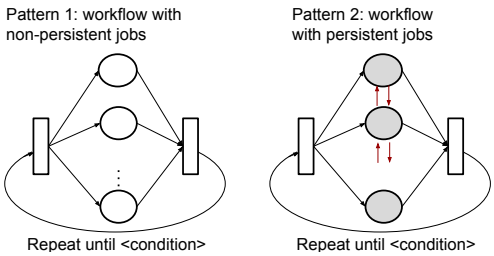

Fig. 1: Skeleton patterns with non persistent and persistent jobs. Circles are nodes that represent slave jobs. Rectangles represent controllers (master jobs). White circles are non persistent jobs while gray ones are persistent jobs. Arrows between persistent jobs represent data transfer and communication.

In the first workflow, jobs are "embarrassingly parallel", i.e. there is no communication between them. The checking convergence process is launched each time a job finishes. In the second workflow, jobs are considered as daemons (persistent jobs). In this case, jobs keep running on the resources, communicate with their neighbours (exchange boundary data) and deliver intermediate computation results. Each job, (i) receives some input data and delivers results, (ii) updates and processes its internal data. The non-reception of the input data is tolerated, i.e. step (ii) can be performed even if the input data are not available. This delays the convergence time.

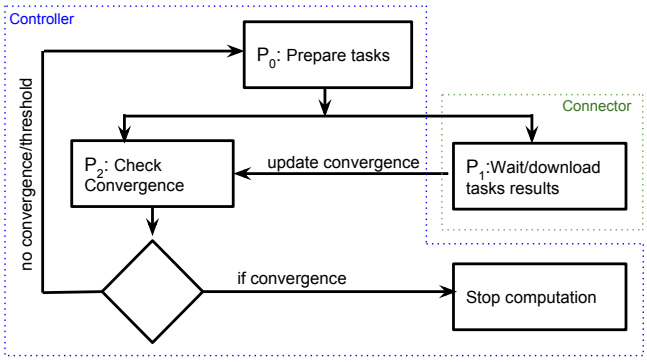

Fig. 2: Decision process in a DW applications

The convergence condition of such dynamic workflows is modelled by a set of processes running on the end-user program as depicted in Fig. 2. In the first step, a process P_0 prepares a given number of jobs. Then, two parallel processes P_1 and P_2 are launched. Process P_1 waits for the slave jobs to finish in order to download their results. Process P_2 periodically checks the convergence condition of the computation. If the convergence is not reached, process P_2 prepares n additional jobs to be submitted (assuming that the number of running jobs is less than a given threshold *ts*). When a job is finished, process P_1 notifies P_2 and the convergence verification is re-launched. Note that P_1 is tightly linked to the infrastructure on which the jobs are running.

The objective of this model is to simplify the implementation of DW e-science applications as defined in section I. This model distinguishes two types of nodes within the same workflow: computing (slaves) and controller (masters) nodes. Computing jobs are submitted to remote resources. A *controller* node is a process that handles computing job submissions and implements the two processes P_0 and P_2. The process P_1 is illustrated through the *connector* component which handles submission, wait and local download of jobs' output files.

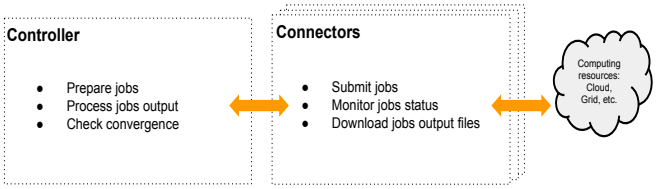

Fig. 3: Computation workflow with convergence condition.

The global architecture of the whole system is illustrated in Figure 3. In what follows, sub section A (resp. sub section B) presents the detailed architecture of the *controller* and *connector* components when jobs are non persistent (resp. persistent).

A. Pattern 1: Skeleton for handling non persistent jobs

An application performing parameters sweeping with convergence condition is an example of our first pattern, pattern 1 (Fig.4). In this pattern, the first step consists in preparing a list of jobs (with their given input files and parameters) to submit. These jobs are inserted into a job queue. This operation is specific to the application and should be implemented by the programmer.

The *connector* module submits jobs to the targeted computing platform, monitors their status and retrieves their output files. The coordination between the *controller* and the *connector* modules is as follows. The *controller* takes care of checking the convergence during all the computation whereas the *connector* performs the runtime execution on the target computing platforms. The *controller* module receives a message from all *connector*s notifying it that a set of jobs have finished and their output files, if any, were downloaded locally. It then proceeds to post-process the output files and checks the convergence condition. If the convergence condition is not reached (and assuming that the number of running jobs is lower than a given threshold), the *controller* prepares a batch of jobs and insert them into the job queue. Otherwise, the *controller* terminates the computation, cleans the job queue and notifies the *connector*s.

A DW application with non persistent jobs can be modeled by this expression

$$W = (c, J, X)$$

where c is a *controller*, J is a set of k jobs and X is a set of n *connector*s.

According to the specification of pattern 1, the code of the DW application can be expressed in object oriented style as follows:

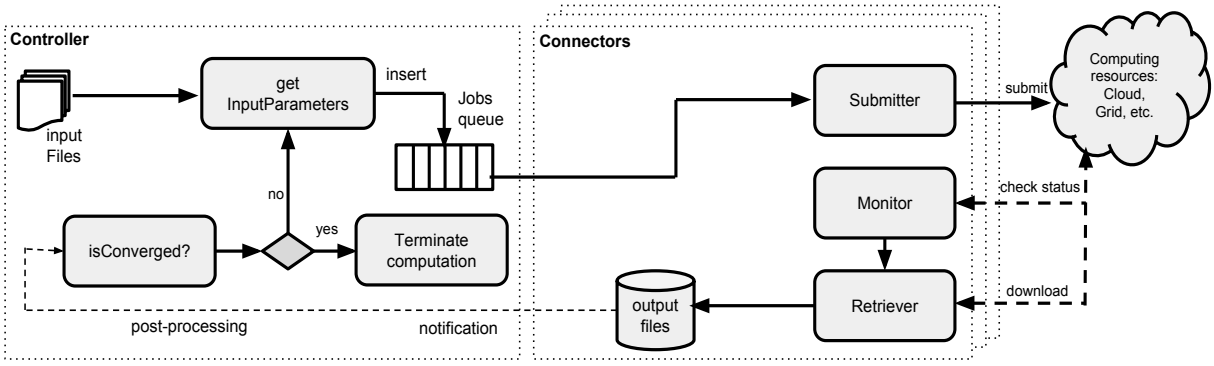

Fig. 4: Computation workflow of pattern 1

```
1  Nodes jobs=new Nodes("jobs:1:k");
2  jobs.setPrecedence(new BusinessPrecedence());
3  Workflow wk=new Workflow(jobs);
4  Controller ctr=new EndUserController(user_parameters);
5  Connector con1= new Infrastructure_X1_Connector(userCredentialsX1);
6  Connector con2= new Infrastructure_X2_Connector(userCredentialsX2);
7  ctr.setConnectorsList(new ArrayList().add(con1).add(con2));
8  wk.setController(ctr);
9  wk.setDispatcher(new BusinessSplitProcess(jobs, wk.getConnectors()));
10 wk.run();
```

The jobs $\{job_i\}_{1 \leq i \leq k}$ composing the workflow are declared in line-1. The initial number of jobs in the workflow can be extremely large and thus complex to define in the code. A method to reduce the definition of jobs is to assign a range of numbers to jobs as shown in line-1. Moreover, dependencies between jobs can be expressed using a *BusinessPrecedence* object in the method *setPrecedence* as shown in line-2. This method is abstract method that can be implemented. If there is no precedence, this line is not required. Line-3 creates an object to handle the dynamic workflow application. A *controller* and *connector* components are instantiated in lines 4-5-6. The former implements processes P_0 and P_2, i.e., checks the convergence and takes the decision of whether running additional jobs. The latter submits jobs to the targeted computing infrastructures $X1$ and $X2$ (such as Microsoft Azure, Amazon EC2, Grid and/or volunteer platforms, etc.) and implements the process P_1, i.e., the handling of the jobs' progress and the download of their outputs. Line-7 associates the two *connector* components to the *controller* $ctrl$ which can handle the computation (line-8). Line-9 dispatches jobs among targeted infrastructures (connectors) according to a predefined policy : available resources, minimization of execution cost (in case of Cloud), minimization of execution time, etc.

It is worth noticing, that the *controller* is a component that should be implemented by the programmer since it is specific to the application, whereas the *connector* depends on the resource providers and is implemented independently of the application.

In order to implement a *controller* class for a given application, the programmer should extend a predefined *Controller* class and implement the following abstract methods:

- getInputParameters(i): This method returns the input parameters of the next job i to be submitted: job name, executable (binary), input files, command line, output files, final (a boolean variable which indicates if the job output must be downloaded)
- isConverged(): it returns a boolean status of the computation convergence condition.
- getfInitialJobsNumber(): it fixes the initial number of jobs to submit when starting the computation.
- getSubmissionThreshold(): it returns the threshold value "ts".
- getAdditionalJobsNumber(): it fixes the number of additional jobs to submit in case the convergence is not reached and the threshold is verified.

The predefined *Controller* class has two methods *initialize()* and *run()* methods. The *initialize()* method starts the computation by sending the initial jobs. The *run()* method is periodically activated and uses the *Connector* component as described in the algorithm 1.

Moreover, in the class *BusinessSplitProcess*, a method called *getJobDispatching()* which should be implemented by the programmer, returns a mapping from the set of jobs J and the *connector*s X.

Algorithm 1: run() method of the *Controller*

```
Data:
Connector[] consList;
1 begin
2     if (not isConverged()) then
3         for i ← 1 to getAdditionalJobsNumber() do
4             Connector con=getConnector(i);
5             con.submitJob(getInputParameters(i));
6     else
7         for i ← 1 to consList.size() do
8             consList[i].stopComputation();
```

In order to implement a *Connector* class for a given infrastructure, the programmer must extend a predefined class and implement the abstract method *postProcessing(Job j)*. The predefined *Connector* class has the following methods:

- submitJob(Job j): submits a job to the target computing platform.

- getRunningJobs(): returns a list of running jobs.
- getFinishedJobs(): return a list of finished jobs.
- stopComputation(): stop all running jobs and clean up the execution environment.
- postProcessing(Job j): a method which processes a finished job referenced by the parameter `j` and notifies the *controller* to verify the convergence condition.
- run(): verifies periodically the submitted job status and calls the *postProcessing* method each time a job is finished.

B. Pattern 2: Skeleton for handling persistent jobs

A DW application with persistent jobs can be modeled by this expression

$$WP = (pc, J, X)$$

where pc is a controller of a set of k persistent jobs J and X represents a set of n *connector*s. Note that unlike the first pattern, k is fixed by the programmer. A DW application with persistent jobs can be expressed in an object oriented paradigm as follows:

```
1 Nodes jobs=new Nodes("jobs:1:k");
2 Workflow wps=new Workflow(jobs);
3 PController ctr=new EndUserPController(user_parameters);
4 Connector con1= new Infrastructure_X1_Connector(userCredentialsX1);
5 Connector con2= new Infrastructure_X2_Connector(userCredentialsX2);
6 ctr.setConnectorsList(new ArrayList().add(con1).add(con2));
7 wps.setController(ctr);
8 wps.setDispatcher(new BusinessSplitProcess(jobs,
        wps.getConnectors()));
9 wps.run();
```

The computation steps are as follows:

1) the problem domain is split into several blocks. Each block will be assigned to one persistent job.
2) Once started, each persistent job deploys a proxy interface to communicate (exchange boundary data) with its neighboring jobs.
3) Each persistent job receives boundary data, updates its block and sends its boundary data back to its neighboring jobs. It asynchronously processes its data block even if no data is received.
4) All intermediate results are merged and the convergence of the whole domain is checked.
5) If the convergence is not reached, go to step 3). Otherwise, all persistent jobs receive a notification to clean and terminate the computation.

As explained in these steps, the persistence aspect often implies communications between neighboring jobs. Hence, the skeleton pattern should propose an efficient communication pattern.

Because persistent jobs are running on a distributed infrastructure, synchronizing the communication between them is not straightforward. To this end, a well defined paradigm to exchange data is needed. This is achieved through *Messages*. In its simple form, a *Message* is an object that encapsulate information related to the sender (a job identifier), the destination (a job identifier), a message tag and a body. The latter can simply be a file, an URL or an embedded data.

Furthermore, exchanging messages have to be done through a uniform communication interface which should be written by the programmer. We therefore consider a *Communicator* class with two methods *receive* and *deliver*. This class is implemented within the persistent job. The *receive* method feeds the persistent job with a list of *Messages* while the *deliver* method delivers a list of *Messages* that are retrieved by the *controller* and routed to their corresponding destination.

Communication between the *controller* and persistent jobs is depicted in Fig. 5. The *controller* receives two types of *Messages* from persistent jobs: *Messages* destined to persistent jobs and *Messages* used by the controller to check the global convergence of the computation. *Messages* are inserted into the corresponding queues. There are k+1 queues : one for each persistent job and one for the global convergence *Messages*. Another process retrieves the *Messages* from jobs queues, prepares the jobs and submit them to the remote resources on which the corresponding persistent job is running.

Messages received by a persistent job are used to accelerate the computation whereas those destined to the *controller* are used to check the global convergence of the computation.

Reception of *Messages* and jobs submission can be executed in parallel. To shield the programmer from handling the synchronization between all these processes, the application controller object must be extended using a *PController* abstract class and the following abstract methods must be implemented:

- getPersistentJobs(): splits the problem domain into several blocks and returns a list of *Job* objects to be run on the remote resources. It is worth reminding that the binary file of each persistent job must implement the *Communicator* class which will be run in the background on the remote resources.
- isConverged(): reads all *Messages* destined to the *controller* and returns the status of the convergence condition as a boolean value.
- getIterationsThreshold(): fixes a maximum value *ts* for the number of iterations allowed in case the convergence has not been reached yet.

The predefined class *PController* has additional methods, namely, *initialize()* and *run()*. *initialize()* calls the getPersistentJobs() method, submits the list of persistent jobs to the remote resources (using the corresponding connectors) and waits for their deployment. Once finished, it retrieves the output of each persistent job.

The *run()* method is activated periodically to check for the convergence, it reads the queues, submits the jobs and retrieves their output (*Messages*) as described in the pseudo-code of the algorithm 2. The convergence is checked in line 2. If there is no convergence, the list of messages for each queue is selected (lines 5-6) in order to be sent to the corresponding persistent job. The method *prepareJobToPSJobs()* prepares a job (lines 8-9) with the "right" communication interface parameter. The *connector* object sends the job to the same computing resource where the persistent job is running (line 10). Note that the

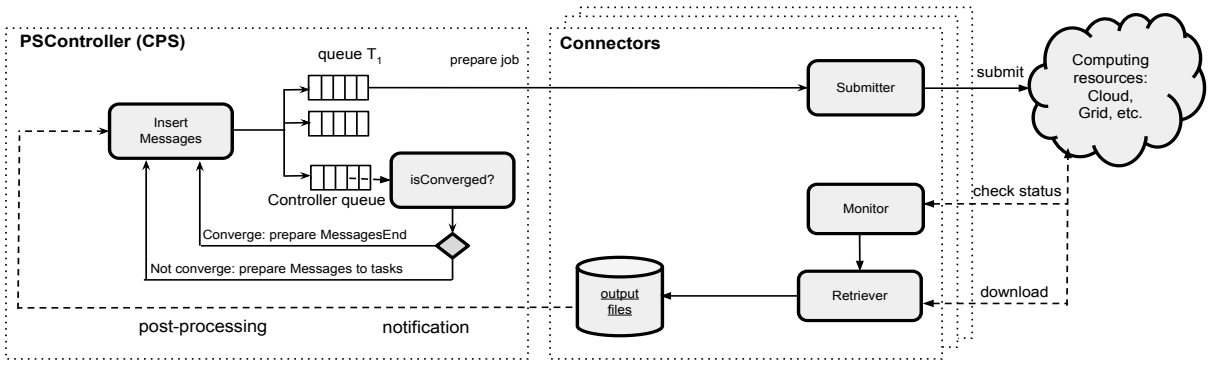

Fig. 5: Computation workflow of pattern 2

Algorithm 2: run() method of the *PSController* class.

```
   Data:
   Queue[] queuesList;
1  begin
2      if (not isConverged()) then
3          for i ← 1 to queuesList.size()) do
4              Queue q←queuesList[i];
5              List<Messages>listMsgs =q.element();
6              int receiver = listMsgs.get(0)).getReceiver();
7              Job Tps←getCorreponsdingJob(receiver);
8              Job j←prepareJobToPSJobs(listMsgs,Tps);
9              Connector con←Tps.getConnector();
10             con.submitOnSameResource(j, Tps);
11     else
12         stopComputation();
```

submission of jobs is not a blocking operation. In case the convergence is reached, the computation is terminated by calling the method *stopComputation()*.

IV. Skeleton implementation for MetaPIGA and NeuroWeb applications

In this section, we propose to apply the two programming paradigms detailed in section III to two DW e-science applications : MetaPIGA and NeuroWeb.

MetaPIGA [24] is a genetic application developed by the Laboratory of Artificial and Natural Evolution (LANE) at University of Geneva. It is a robust implementation of several stochastic heuristics for large phylogeny inference. The objective of MetaPIGA consists in finding the most suitable individual, i.e., the phylogenetic tree that has the greatest probability (called also the greatest Likelihood) of generating an observed sequence of genes for different living species. We start from an input dataset which is a file containing an alignment of DNA (nucleotides A, C, G, T) sequences from different living species. Then, the client program (MetaPIGA) generates R independent analyses. Each analysis consists in generating P independent random populations of trees, which will undergo mutation and selection phases. At the end, several solution trees generated from all analyses are merged into a consensus tree. Note that we cannot know in advance the number of analyses R needed to obtain the final solution: MetaPIGA will continue generating analyses and "improving" the consensus tree until the process converges. The execution time of each simulation depends on the dataset size, computing resources and runtime results of the execution.

The second application NeuroWeb [20] is developed by The University of Applied Sciences, Western Switzerland, builds neuronal maps of brain activity with non-invasive measurements. For instance, it can be used to identify sources of brain "abnormal" activities such as epileptic crisis, Parkinson, Alzheimer, etc., for remote surgery. It "extracts" the activity of different regions of neurons from captors attached on the head of the patient. The estimated images of brain activity are a solution to a high-dimensional, non-differentiable optimization problem. The solution space size is extraordinarily large, about 10^7. NeuroWeb is assumed to produce a dynamic neural map (*DNM*) in which each row represents the electromagnetic activity of one region of neurons.

The application is an iterative process which starts by choosing a random map called DNM^0. At each iteration i, a new matrix DNM^i is constructed based upon DNM^{i-1}. Iterations stop when, the DNM^i matrices can no longer be improved, meaning that the convergence is reached. The distribution version of NeurWeb consists in splitting the DNM matrix into several blocks B_j. Each block is processed by a daemon process, called "persistent server" (*PS*), which executes an iterative asynchronous algorithm. During its execution, a *PS* receives boundary data from its *PS*s neighbors. Note that the processing of a block *B* by a *PS* can take place even if no input data are received from the neighbors of *PS*. Indeed, *PS* continues to improve its block *B* even if it does not receive any data from its neighbors.

The first paradigm will be used for MetaPIGA (non persistent jobs) and the second one for NeuroWeb (persistent jobs).

A. The MetaPIGA use case

Listing 1 shows the pseudo-code of the "MetaPIGAController", the *controller* class of the MetaPIGA application. Note that although there are portions of source codes which are not shown here, this pseudo-code includes all relevant details about the structure of the class.

Listing 1: The Java class of The MetaPIGA *controller*

```
1  class MetaPIGAController extends Controller{
2    double MIN_SCORE=0.01; int counter=0;
3    public long getInitialJobsNumber(){return 300;}
4    public long getsubmissionThreshold(){return 10000; }
5    public getAdditionalJobsNumber(){return 100; }
6
7    public Job getInputParameter(int i){
8      Job param= new Job();
9      param.jobName="T"+i;
10     param.executable="metapiga2.jar";
11     String inputFile=createInputFile(i);
12     param.cmdLine="java -jar metapiga2.jar noupdate" +"silent
           "+inputFile+" > metapigaout.log 2>&1";
13     param.compressedOutPutFile="output.nex.zip";
14     param.isFinal=false;
15     return param;
16   }
17
18   public boolean isConverged(){..}
19 }
```

A minimal effort is required to implement an application-specific *controller* class. Basically, a small effort is needed for both *getInputParameter(i)* and *isConverged()* methods as shown in Listing 1. Technically, preparing jobs consists in creating a different input file for the same *metapiga2.jar* binary file (line-12). Then, the output result for each input file will be compressed in *output.nex.zip*, downloaded by the *connector* and processed in order to generate a phylogenetic tree. Regarding the computing infrastructure, the programmer has the option to either write their own *connector* or select one from the predefined *connector*s classes such as *AzureConnector* for Microsoft Azure or *AmazonEC2Connector* for Amazon EC2. Writing a *connector* class requires mainly re-implementing the interface of the modules "submitter, watcher and retriever" (Fig. 4) since the scheduling back-end of jobs is hidden to the programmer.

The code of the MetaPIGA computation can be written in few lines of Java code. Note that in the line-1 of Listing 2, the jobs range should be compliant to what the *getInitialJobsNumber()*, *getsubmissionThreshold()* and *getAdditionalJobsNumber()* methods return. As shown, the total number of tasks should not exceed the threshold 200. Moreover, there is no precedence rules between jobs.

Listing 2: MetaPIGA main code using our model

```
1  Nodes jobs=new Nodes("T:0:200");
2  jobs.setPrecedence(new DefaultBusinessPrecedence());
3  Workflow wk=new Workflow(jobs);
4  Controller ctr=new MetaPIGAController(user_parameters);
5  Connector con1= new AzureCloudConnector(azureUserCredentials);
6  Connector con2= new AmazonCloudConnector(EC2UserCredentials);
7  ctr.setConnectorsList(new ArrayList().add(con1).add(con2));
8  wk.setController(ctr);
9  wk.setDispatcher(new DefaultBusinessSplitProcess(jobs,
       wk.getConnectors()));
10 wk.run();
```

B. The NeuroWeb use case

The main code of the NeuroWeb application is similar to the one shown in Listing 2 except for line-4 where the programmer should instantiate *PController* object *PController ctr=new NeuroWebPController(user_parameters)* in order to handle the persistent aspect of the computational workflow of the NeuroWeb application.

Listing 3 shows the pseudocode of the *PController* class of the NeuroWeb application. The method *initialize()* creates the persistent jobs and retrieves the information related to their communicator interface. Depending on the available resources and user *BusinessSplitProcess* strategy, the submission is handled by a list of connectors and each job is assigned a connector.

Listing 3: Java class of the NeuroWeb controller

```
1  class NeuroWebController extends PController{
2
3    public List<Jobs> getPersistentTasks(List<Messages>
         msgsFromMaster) {
4      this.splitDomain(nbMaxTask, msgsFromMaster);
5      nbMaxTask = msgsFromMaster.size();
6      List<Jobs> listjobs = new ArrayList<Job>();
7      for (int i = 0; i < nbMaxTask; ++i) {
8        Job j = new Job();
9        j.jobName = "T" + i;
10       j.executable = "pstaskbinary.jar";
11       Directives directiveClient = this.getDirectivesToTask(i);
12       String taskInputzip = util.serialize(listmessagetoworker,
             directiveClient);
13       j.inputFile = taskInputzip;
14       j.compressedOutPutFile = "serializedMessages_" + i + ".zip";
15       j.cmdLine = "nohup java -cp .:* neuroweb.pstask " + i + " "
             + nbMaxTask + " " + j.compressedOutPutFile +" "
             + taskInputzip;
16       param.isFinal = false;
17       listjobs.add(j);
18     }
19     return listjobs;
20   }
21
22   public void initialize() {
23     try {
24       List<Messages> msgsFromMaster= new
             ArrayList<Messages>();
25       List<Jobs> psTasksList=getPersistentTasks(msgsFromMaster);
26       for (Job psJob: psTasksList){
27         Connector con = getConnectorForJob(psJob);
28         psJob.setConnector(con);//set the job's connector
29         con.submitJob(psJob);
30       }
31       waitForJobs(psTasksList);//waiting all Workers jobs
32       for (Job psJob: psTasksList){
33         Port p=getCommunicationJob(psJob.compressedOutPutFile);
34         psJob.setPort(p);
35       }
36   }}
```

In Listing 4, the *isConverged()* method reads all messages destined to the *controller* and decides whether or not to continue the computation. If the convergence is not reached, this method populates the queues of the persistent jobs with *Messages* objects containing the required information needed by the *controller*. Otherwise, the *controller* prepares a *MessageEnd* object for each persistent job to inform it that the convergence is reached. If a persistent job receives a *MessageEnd* object, its stops and cleans up the computation.

Listing 4: run() method in the NeuroWeb *controller* class

```
1  public void run(){
2    if ( ! isConverged() ){
3      while (queueListMessage.size() > 0) {
4        List<Messages> listMessage = (List)
             queueListMessage.element();
5        int receiver = ((Messages) listMessage.get(0)).getReceiver();
6        Job psTask=getCorrependingJob(receiver);
7        Job j=prepareJobToPsTask(listMessage,psTask);
8        Connector con=psTask.getConnector();
```

```
9           con.submitOnSameResource(j, psTask);
10        }
11      }else{stopComputation();}
12    }
```

V. Experiments and comparisons

The objective of this section is to evaluate the performance of the two proposed programming paradigms. In the context of our study, "performance" covers:

- the ease of writing and clarity of the produced source code,
- the time overhead generated by the proposed models.

Two experiments were conducted. They concerned the MetaPIGA application described in section IV.

The first experiment was to write and deploy (on several platforms) MetaPIGA application, using the paradigm described in section III. The goal is to show that the proposed paradigms support two main features : portability (write once, deploy many) and interoperability (deploy the same application on several heterogeneous distributed platforms at the same time).

The second experiment was to compare the overhead generated by our model and that generated by another "concurrent" tool : GC3Pie. Finally, a comparison is made between our model and GC3Pie regarding the clarity, the intuitiveness and the conciseness of the produced source codes.

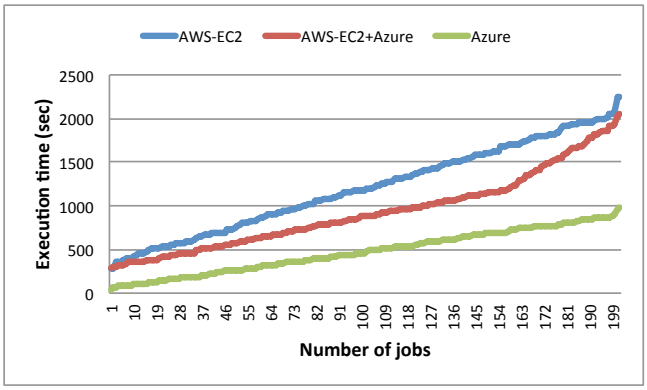

Fig. 6: Execution time of MetaPIGA using the first paradigm. The used cloud platforms are Amazon EC2 and Microsoft Azure.

In both experiments, a set of 200 MetaPIGA jobs, programmed and deployed with our model, run on both Amazon EC2 (AWS-EC2) and Microsoft Azure cloud platforms. In these experiments, the convergence condition is simply reached once the number of execution jobs exceeds 200. We used 20 virtual machines (VMs) based on the *Small* computing Amazon and Azure instance type (see Table I).

The first simulation consists of three scenarios. Scenario 1 (resp. scenario 2) uses 20 VMs of Amazon EC2 (resp. Microsoft Azure) resources. Scenario 3 uses heterogeneous resources from both platforms (10 VMs each). The developed program run on the user machine and undertook the following operations:

- start the VMs on the cloud platforms,
- upload the input data of each job from the user machine to the corresponding VM,
- run remotely MetaPIGA job on the VMs,
- wait and download the output result from the VM to the user machine.

TABLE I: Characteristics of the VM instance type.

Platform	VM type	Memory	CPU	OS	Location
AWS-EC2	m1.small	1.7 GB	Intel-Xeon @2.00GHz	Ubuntu	North Europe
Azure	small (A1)	1.7 GB	Intel-Xeon @2.20GHz	Ubuntu	North Europe

Figure 6 illustrates the execution time of the three scenarios. The execution time on Azure platform shows better performance than Amazon-EC2. This may be explained by the difference in VM CPU speed as shown in Table I. Sharing the same number of jobs on a heterogeneous platform composed of AWS-EC2 and Azure computing resources leads, as expected, to accelerate the execution time compared to AWS-EC2 simulation. This experiment shows that our model is able to support several cloud platforms within the same deployment. It is worth noticing that our classes can be easily extended to other Infrastructure as a Service (IaaS) such as OpenStack and StratusLab [28], [29]. The aim of the next experiment is to compare our two paradigms with the GC3Pie one.

The GC3Pie programming model considers mainly the following classes: *Application*, *SequentialTaskCollection* and *SequentialTaskCollection*. The *Application* class corresponds to a single job and consists of a generic class that programmers should extend to define their jobs. It enables to describe a job at a high level: programmers specifies the list of input and output files, command line to execute, resource requirements and limits, pre-and-post-treatment. Depending on the targeted executing platforms, GC3Pie translates the job description to the adequate format in order to execute it on the corresponding remote resource.

The work-unit in GC3Pie programming model is called a *Task*. An *Application* object is a primary instance of *Task*. A *Task* is a composite: meaning that it can be a single *Task* or a workflow of *Task*s. So, a complex workflow is built by simply composing *Task*s in different ways using the classes *SequentialTaskCollection* and *ParallelTaskCollection*. These two classes are abstract and provide two ways to schedule and execute a collection of *Task*s. The programmer should extend them in order to build a workflow of *Tasks*. The former executes a collection of *Tasks* sequentially. Also, it enables the programmer to perform, after each *Task* execution, a decisional treatment through implementing a given abstract method. A possible decisional treatment can be submitting a new *Task* (which can be a workflow of *Tasks*) of resubmitting a failed one. The latter executes in parallel a collection of independent *Tasks*. By using these two classes one can create a dynamic workflow.

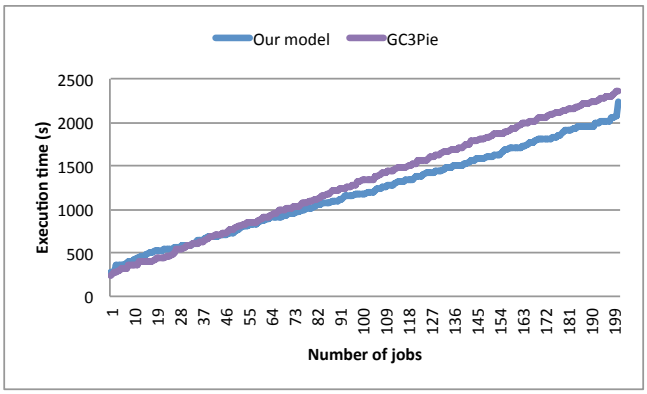

Fig. 7: Execution of MetaPIGA on Amazon EC2: performance comparison with the GC3Pie programming model.

The second experiment considers the same computing resources as in the first experiment (scenario 1). It aims to compare the time overhead of GC3Pie and our programming paradigms. To do so, we mounted a *Slurm* [16] based cloud cluster on Amazon EC2 using the tool elasticluster [30]. This cloud cluster is composed of 20 VMs having the same VMs configuration as Table I. Figure 7 illustrates the performance comparison with the GC3Pie programming model. We can see that the execution time is almost the same. Thus, our model is comparable to GC3Pie in term of computing performance.

TABLE II: Comparison with the GC3Pie programming model.

Criteria	Our model	GC3Pie
Convergence condition	easy to program.	must be expressed at the application level using loop and condition statements.
Workflow	supports a workflow composed of simple tasks.	supports workflows where a task can be also a workflow of tasks.
Cloud IaaS	supports multiple cloud IaaS.	supports multiple cloud IaaS.
Easiness	- the convergence condition is easy to handle. - There is a mechanism for users to split jobs over the available infrastructures.	- the convergence condition is not easy to handle. - the API is more mature and proposes a command line interface to manage the running jobs.
Configuration	does not require any installation. The user only needs to import a library package in her/his project.	- requires installation environment. - infrastructures are described in a configuration file.

The programmability and the easiness of our two programming paradigms is not easy to measure. One of the most straightforward evaluation metric is SLOC (Source lines of code), which consists in counting the number of lines of a program source code. However, this criteria is very subjective and depends on the programmer.

Table II presents a comparison with the GC3Pie though predefined programming criteria. Conceptually, GC3Pie is interesting and provides more flexibility to "build" composite workflow. In addition, it supports running application on HPC systems (clusters). Our model is easier to use and provides a more human-friendly convergence specification, specially for users with no programming experience. Besides, It support master/slave architecture where jobs are persistent.

VI. Conclusion

In this work we presented two high level programming paradigms to develop complex scientific applications and deploy them on large scale distributed and heterogeneous infrastructures. Complex scientific applications are defined as parallel and distributed iterative applications where the number of iterations and jobs per iteration are not known in advance. The two paradigms are syntactically similar. The first paradigm targets scientific applications with non persistent jobs while the second paradigm addresses applications with persistent jobs.

The two proposed paradigms offer a very easy programming model and shield the end-users from the technical complexity of handling job submission and threads. They have the advantage to target more than one computing resources such as cloud or grid platforms. Adding a new type of computing platform is relatively easy and can be done independently of the business logic of the application. The two paradigms, easy to use though they are, require the users to have a very basic knowledge about *Object Oriented Programming* in order to understand and use them.

The proposed paradigms were tested in the concrete cases of two scientific applications: MetaPIGA and NeuroWeb. Deployments were tested on different infrastructures: Amazon, Azure, OpenStack and volunteer computing resources.

Future work will consists on applying these programming paradigms on more scientific applications use-cases and develop a Graphical User Interface supporting the proposed paradigms.

References

[1] M. B. B. Marko Niinimaki, Nabil Abdennadher and D. Racordon, "High-level api for xtremweb-ch: Concurrent computing with a uniform interface," University of Applied Sciences, Western Switzerland, hepia, Tech. Rep., 2013.

[2] R. M. Sergio MAFFIOLETTI and T. Aleksiev, "Computational workflows with gc3pe," in *EGI Community Forum 2012 / EMI Second Technical Conference*, March 2012. [Online]. Available: \url{http://pos.sissa.it/archive/conferences/162/063/EGICF12-EMITC2_063.pdf}

[3] A. Costantini, R. Murri, S. Maffioletti, and A. Laganà, "A grid execution model for computational chemistry applications using the gc3pie framework and the apppot vm environment," in *Proceedings of the 12th International Conference on Computational Science and Its Applications - Volume Part I*, ser. ICCSA'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 401–416. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31125-3_31

[4] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers," *Softw. Pract. Exper.*, vol. 40, no. 12, pp. 1135–1160, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1002/spe.v40:12

[5] M. Leyton and J. Piquer, "Skandium: Multi-core programming with algorithmic skeletons," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, Feb 2010, pp. 289–296.

[6] "Cilk Plus," https://software.intel.com/en-us/intel-cilk-plus.

[7] C. Hélène, L. Minh-Hoang, and L. Sébastien, "Parallelization of Shallow-water Equations with the Algorithmic Skeleton Library SkelGIS ," *Procedia Computer Science*, vol. 18, no. 0, pp. 591 – 600, 2013, 2013 International Conference on Computational Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050913003669

[8] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: Interfaces and implementations," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 247–260. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629600

[9] "Hadoop system," http://hadoop.apache.org/.

[10] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855742

[11] M. Steuwer, P. Kegel, and S. Gorlatch, "Skelcl - a portable skeleton library for high-level gpu programming," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 1176–1182.

[12] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.

[13] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[14] F. Dittmann, "Algorithmic skeletons for the programming of reconfigurable systems," in *Software Technologies for Embedded and Ubiquitous Systems*, ser. Lecture Notes in Computer Science, R. Obermaisser, Y. Nah, P. Puschner, and F. Rammig, Eds. Springer Berlin Heidelberg, 2007, vol. 4761, pp. 358–367. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75664-4_36

[15] "TORQUE," http://www.pbsworks.com/?AspxAutoDetectCookieSupport=1.

[16] "SLURM batch system," http://www.schedmd.com/.

[17] W. Qiang and A. Konstantinov, "Towards cross-middleware authentication and single sign-on for arc grid middleware," *Computer Science - Research and Development*, vol. 23, no. 3-4, pp. 267–274, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00450-009-0084-6

[18] "OpenStack project," http://www.openstack.org/.

[19] M. B. Belgacem, N. Abdennadher, and M. Niinimaki, "Virtual EZ Grid: A Volunteer Computing Infrastructure for Scientific Medical Applications," *International Journal of Handheld Computing Research*, vol. 3(1), pp. 74–85, 2012.

[20] Mohamed Ben Belgacem, Nabil Abdennadher, Marko Niinimaki, "Programming distributed medical applications with XWCH2," in *Proceedings of HealthGrid 2010*, vol. 159, France, June 2010.

[21] N. Abdennadher and R. Boesch, "Deploying phylip phylogenetic package on a large scale distributed system," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, May 2007, pp. 673–678.

[22] D. M. Squire, W. Mueller, H. Mueller, and T. Pun, "Content-based query of image databases, inspirations from text retrieval: inverted files, frequency-based weights and relevance feedback. Technical Report 98.04, Computer Vision Group, Computing Centre, University of Geneva," 1998.

[23] "enviroGRIDS FP-7 project," http://www.envirogrids.net/.

[24] "MetaPIGA 2 - Large phylogeny estimation," http://www.metapiga.org/.

[25] R. Helaers and M. Milinkovitch, "Metapiga v2.0: maximum likelihood large phylogeny estimation using the metapopulation genetic algorithm and other stochastic heuristics," *BMC Bioinformatics*, vol. 11, no. 1, p. 379, 2010. [Online]. Available: http://www.biomedcentral.com/1471-2105/11/379

[26] D. Di, "Histoire du peuplement de l'asie orientale révèlée par le système hla," Ph.D. dissertation, University of Geneva, Switzerland, 03/25 2013. [Online]. Available: http://archive-ouverte.unige.ch/unige:27983

[27] "Soil and Water Assessment Tool," http://swat.tamu.edu/.

[28] "stratusLab project," http://stratuslab.eu/.

[29] *StratusLab Cloud Distribution*. European Research Activities in Cloud Computing, Cambridge Scholars Publishing, 2012, ch. 10.

[30] "Elasticluster web project," http://gc3-uzh-ch.github.io/elasticluster/.