CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3453

JASAG: a gridification tool for agricultural simulation applications

M. Arroqui[1,2,*,†], J. Rodriguez Alvarez[1,2], H. Vazquez[1,3], C. Machado[2], C. Mateos[1,3] and A. Zunino[1,3]

[1]*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Buenos Aires, Argentina*
[2]*Facultad de Ciencias Veterinarias – UNICEN, Pinto 399, CP 7000, Tandil, Buenos Aires, Argentina*
[3]*ISISTAN Research Institute – UNICEN, Pinto 399, CP 7000, Tandil, Buenos Aires, Argentina*

SUMMARY

The Grid Computing paradigm aims to create a 'virtual' and powerful single computer with many distributed resources to solve resource intensive problems. The term 'gridification' involves the process of transforming a conventional application to run in a Grid environment. In that sense, the more automatic this process is, the easier is for developers with low expertise in parallel and distributed computing to take advantage of these resources. To date, many semiautomatic gridifiers were built to support different gridification approaches and application code structures or anatomies. Furthermore, agricultural simulation applications have a particular common anatomy based on biophysical entities, such as animals, crops, and pastures, which are updated by actions, such as growing animals, growing crops, and growing pastures, along simulation execution. However, this anatomy is not fully supported by any of the existing gridifiers. Thus, this paper presents Agricultural Simulation Applications Gridifier (ASAG), a method for easy gridification of agricultural simulation applications, and its Java implementation, named Java ASAG (JASAG). The main design drivers of JASAG are middleware independence, separation of business logic and Grid behavior, and performance increase. An experimental evaluation showing the feasibility of the gridification method and its implementation is also reported, which resulted in speedups of up to 25 by using a real agricultural simulation application.

Received 29 May 2014; Revised 10 November 2014; Accepted 15 November 2014

KEY WORDS: agricultural simulation; Grid Computing; gridification

1. INTRODUCTION

Grid Computing is a computing paradigm that coordinates distributed resources for the purpose of solving problems in interinstitutional contexts [1]. Specifically, Computational Grids are distributed environments for resource-demanding applications [2]. In Computational Grids, the main resource is processing power, followed by storage, network resources, and data, which are combined to form a 'virtual' and powerful super computer.

Ideally, Grid Computing should provide users with a simple way to implement and deploy their applications in a Computational Grid, or from now on Grid. Thus, researchers have been working on *gridification methods* to make the task of Grid-enabled applications easier. However, the deployment of any application in a Grid by using a single gridification method has not been achieved yet because of the intrinsic variety of application structures (or anatomies [3]), functionalities, and technologies [4]. Therefore, the topic is subject of active research in the area.

Most gridification methods, materialized later as tools, were developed either to help developers to build Grid applications or migrate their sequential applications. These tools can be divided

*Correspondence to: Mauricio Arroqui, Facultad de Ciencias Veterinarias, UNICEN, Pinto 399, CP 7000, Tandil, Buenos Aires, Argentina.
†E-mail: marroqui@exa.unicen.edu.ar

into two groups according to how they materialize the gridification process. The first group is based on the Application Programing Interface (API) concept, which in this context represents the communication interface between the application business logic and the Grid middleware. Examples of such APIs are JavaSymphony [5], JavaGCL [6], and Java CoG Kit [7]. The weak point of these kinds of tools is that calls to the Grid API are unavoidably merged into the business logic code. On the other hand, the second group is composed of more recent tools that materialize semi automatic transformation methods to obtain the Grid counterparts of sequential applications [3, 8]. Examples are Java Generalized Reactive Intelligent Mobility (JGRIM) [9] and EasyFJP [10]. This kind of tools allows users to Grid-enable applications with low developer intervention, and in many cases, without merging the business logic code with Grid-specific code.

To date, a number of Grid technologies and gridification methods have been applied to CPU-intensive simulation applications [11–13]. Particular types of simulation are those from the agricultural domain, such as agricultural production systems simulator (APSIM) [13] and Simugan [14]. Agricultural simulation applications are used to simulate many factors such as crop and livestock yields, soil organic carbon content, greenhouse gas emissions, energy balance, among others [15]. Additionally, this type of simulation is climate-driven – affecting pasture growths – and is subject to market uncertainties, which yield different economic outcomes [16]. Thus, the experimentation with agricultural models to reach accurate results makes most agricultural simulation applications big CPU consumers.

Agricultural simulation applications have a common application structure because of the ways their entities (e.g., animal, soil, pasture, and cash crop) and tasks/computations (such as feeding animals, growing animals, moving animals, growing pasture, and growing crops) interact during a simulation. For example, when a rule-driven task is active in a livestock simulation (i.e., animal-feeding task with decision capability to increase or decrease pasture allowance to cattle) and then a particular *condition* is met (lower animal live weight than expected in a given simulation time frame), the task execution flow may unexpectedly change because of the triggering of additional tasks. This is the case when for example extra feeding is not possible within the same feeding paddock, so a move-animal-to-the-following-paddock task activates and runs. Moreover, Jones *et al*. [17] have pointed out that agricultural simulation applications have a similar modular structure from an architectural perspective. This similarity is evidenced in the entity–task relationships mentioned earlier, where entities represent parts of the biophysical system (e.g., a farm) and tasks represent the actions that modify and operate on these entities. In addition, certain rule-driven tasks are conditionally applied depending on simulation status, creating new task activation chains. Consequently, many task execution flows are implicit because of the modifications in the tasks execution order. Then, the common architecture for agricultural simulation applications combines workflow structures together with event-based elements.

Therefore, this essential similarity (i.e., common architecture) could be exploited by a gridifier as a common application anatomy that could be parallelized to improve overall simulation performance. In this line, many ad hoc Grid developments for agricultural simulation applications have indeed arisen in past years. For example, APSIM [13] is an agricultural simulation tool that was gridified [15] in order to decrease the execution time of a particular experiment. This experiment, which consisted of 325 scenarios of 122-year simulation each, could have taken 30 years in one computer in its sequential form, but after the gridification, the experiment took 10 days. The gridification of APSIM was carried out by using an API-oriented tool, so the Grid code was spread within the business logic code. For that reason, expert developers were needed to implement, test, and maintain the code [3].

Moreover, semiautomatic gridifiers, such as CAMELotGrid [18] and EasyFJP [10], and those described by Mateos *et al*. [3] avoid Grid code intrusion in the business logic code, which in turn require less user expertise, among other benefits. Because of their code structure, however, these tools cannot completely satisfy the parallelization opportunities offered by agricultural simulation applications. For instance, CAMELotGrid and EasyFJP were designed to gridify cellular automata and divide and conquer applications, respectively. As we will explain later, it is very expensive and even impractical to adapt an agricultural simulation application to the anatomy supported by existing tools.

DOI: 10.1002/cpe

This article introduces a new gridification method, called Agricultural Simulation Applications Gridifier (ASAG), in order to support the gridification of agricultural simulation applications. This method aims to exploit parallel opportunities of agricultural simulation applications taking into account inherent characteristics such as interactions between entities and tasks, and implicit execution flows. This method also promotes the benefits of traditional gridifiers; namely, separation of business logic and Grid behavior, performance increase, and low expertise in parallel computing by users. This latter feature is very important in disciplinary domains such as agriculture, where users should be more focused on modeling and performing simulations rather than handling Grid technologies to scale out their applications. The gridification method implementation, called Java Agricultural Simulation Applications Gridifier (JASAG), was implemented by using the Java programming language to operate cluster environments. This was carried out in order to test the viability of the proposed gridification approach.

The remainder of this article is organized as follows. Section 2 discusses related gridifiers. Section 3 details the proposed gridification method. Section 4 describes the implementation of the new gridification method in a software tool. Section 5 presents the results and evaluation of the gridification method with a real agricultural simulator called Simugan [14]. Finally, Section 6 presents the conclusions and future works.

2. RELATED WORK

Broadly, the main objective of semiautomatic gridifiers is to ease the task of Grid-enabling applications. These tools are designed to keep developers' focus on business logic implementation and testing [3] as much as possible, rather than dealing with Grid programming. After implementation, the code is gridified to turn in into Grid-enabled code. For this purpose, a variety of gridifiers were developed in order to allow users to quickly Grid-enable different types of applications. The main difference across gridifiers lies in certain known aspects [3], namely application code anatomy, gridification granularity, code intrusiveness, developer usage limitations, and middleware support, which are explained later in the context of the most relevant related work. From now on, by 'gridifier', we mean tools promoting sequential code transformation methods and not API-based gridifiers.

2.1. Anatomy

In order to be successfully gridified, an application that has to be run on a Grid environment should obey an anatomy [3]. In that sense, the pure workflow anatomy prescribed by DAGman [19, 20], Pegasus [21], Triana [22], Taverna [23], GridFlow [24], g-Eclipse Workflow Builder [25], and Kepler [26] is commonplace in the literature, and it is mostly used in scientific applications. With the workflow anatomy, the application functionality is represented as an ordered task set, where the order is manifested as task dependencies. Additionally, dependent tasks can also use the *data flow* [27] anatomy, which allows dependent tasks to execute in parallel to gradually process data right after nondependent tasks, such as Unix pipes [28]. In that sense, Askalon [29] supports the data flow anatomy only, while Triana, Kepler, and Java CoG Kit enhance the workflow anatomy with data flows.

The divide and conquer code anatomy, alternatively, uses recursion to hierarchically divide a problem into many small problems (i.e., tasks) that can be resolved in parallel. Once each small problem has been resolved, the overall problem result is derived from each small problem result. This anatomy is supported by recent tools such as Sync Generator [30], EasyFJP [10], and BYtecode Gridifier (BYG) [31]. BYG can also be configured to support bag of task (BoT) anatomy. The BoT anatomy is presented as a set of fully independent tasks that can be run in parallel and are controlled by a single master task.

The component-oriented anatomy is based on reusable software pieces that are compounded in a loosely coupled fashion to form a larger system. This anatomy is found in JGRIM [9], where specific components from the original sequential application are transformed into Mobile Grid Services (MGS) whose functionalities are exposed via Web services. After transformation, each MGS is able to interact with other MGSs and indirectly use middleware-level Grid services such as resource management, mobility, and thread execution.

DOI: 10.1002/cpe

Another anatomy is cellular automatas (CA), which is used by CAMELotGrid [18]. A CA consists of a finite matrix of state cells where each cell has neighbor cells. An initial state is selected by assigning a state to each cell. The CA evolves in discrete steps where the state of each cell and its neighbors is changed according to mathematical models. CAMELotGrid supports CAs in a simulation environment where natural system applications are modeled by a large number of simple elements (the cells of the automata) with local interaction only (the neighbors).

Lastly, Grid Execution Management for Legacy Code Architecture (GEMLCA) [32], Gridify and Running Applications on Service-oriented Grids (GRASG) [33], and GriddLeS [34] use binary codes to gridify applications. In that sense, these tools wrap the binary code of an application with a single-threaded Grid application in a black-box fashion. Thus, these kinds of tools do not define a particular code anatomy and are ideally used in closed execution environments where source codes are compiled to specific hardware platforms and operating systems.

2.2. Gridifier granularity

Gridifier granularity is a qualitative metric that attempts to measure the amount of functionality a gridified piece of software has [3]. Basically, when Grid-enabled, these pieces of software are associated execution units (jobs or tasks) for which the Grid directly provides scheduling and execution services. Despite gridifier granularity takes continuous values, it is usually discretized for practical reasons as heavy weight, medium weight, and light weight. Thus, *heavy weights* are those pieces of software with a great amount of functionality, whereas *light weights* are those providing little functionality.

Tools that generate heavy-weight gridified pieces of code such as GEMLCA, GRASG, and GriddLeS are characterized by the gridification of the whole input binary code. This means that the application is seen as a 'black box' from the tools point of view. After gridification, parallelization opportunities are limited to running multiple instances of the entire Grid-enabled application in parallel.

Tools that generate medium-weight gridified pieces of code are mostly those that follow the workflow approach, the BoT anatomy, and component oriented. First, under pure workflow-based tools such as the ones mentioned earlier, applications comprise many dependent tasks with specific functionality, which means that each task will start its execution when the tasks on which it depends on have finished. Second, similar to workflows, under the BoTs anatomy (a configurable option of the BYG tool), tasks are independent from each other and can run in parallel if enough computational resources are available. Lastly, in component-oriented tools such as JGRIM, applications consist of many components where each component is independent from each other as well, so they can run in parallel.

Finally, tools that generate light-weight gridified pieces of code are those supporting the divide and conquer paradigm, workflows that support data flow communication between dependent tasks, and tools that use the CA anatomy. Tools with the divide and conquer support, such as Sync Generator, EasyFJP, and BYG, support parallelism inside Java methods. This means that individual methods or parts thereof in the application to gridify are mapped to different execution units in the Grid-enabled application. Moreover, workflow-based tools such as Triana, Askalon, and Kepler allow two dependent tasks to run concurrently but at the same time communicating data via a producer–consumer data interchange strategy. Finally, tools using the CA anatomy, such as CAMELotGrid, enable for automata division into many smaller automatas that can be processed in parallel.

2.3. Intrusiveness

Intrusiveness indicates to what extent the Grid-aware code is merged with the business logic application code after the gridification process has been applied [3]. In that sense, GEMLCA, GRASG, GriddLeS, EasyFJP, BYG, JGRIM, CAMELotGrid, and most of the workflows mentioned earlier do not merge the Grid-aware code with the logic code. GEMLCA, GRASG, and GriddLes operate based on binary codes, so the source code is not modified at all. Furthermore, JGRIM and EasyFJP use dependency injection [35], a mechanism for separation of concerns very popular in Web applications, to avoid merging grid-related code with the application logic. Moreover, BYG modifies

DOI: 10.1002/cpe

Java bytecodes, acting as a post-processor after the Java compiler. Under CAMELotGrid, an input automata is automatically divided at runtime depending on the computational resources needed, but the automata described by the user in the application code is not altered. Furthermore, most workflows are modeled by users without taking into account Grid programming details because these tools provide transparent methods to gridify workflows without changing their code.

On the other hand, tools that somewhat merge the Grid code with the application logic code are Sync Generator and some workflows tools such as those of Taverna and Kepler, which require the user to define the Grid node where high resource consumer tasks are to be executed. Sync Generator needs that all parallel candidate methods be manually annotated in the application code prior to gridification.

2.4. Application restrictions

Even when gridifiers aim to fully isolate users from Grid programming details and source code modifications in the input application, users should keep in mind certain implementation restrictions before starting to use some tools in case they want to gridify their application effectively. GEMLCA, GRASG, and GriddLeS are restriction-free because they gridify the whole application binary code, and thus, source code modifications do not apply.

However, if developers use Sync Generator, JGRIM, EasyFJP, or BYG, the whole application should follow some Java code conventions so that the gridification process works correctly. Java conventions to follow are Serialization, plus the JavaBeans design pattern [36] (with the exception of Sync Generator). This does not mean to merge business logic code and Grid code but to slightly modify the sequential application code so as to obey these source code conventions. For example, in BYG, Java classes that will be interchanged between the Grid nodes have to implement the *java.io.Serializable* interface so that computations can be deployed on nodes where these classes are not initially available.

On the other hand, CAMELotGrid and the workflows mentioned earlier are more restrictive because the whole application code should respect the anatomy supported by each gridifier. These tools provide an integrated development environment where the user defines cells and mathematical functions in the case of CAMELotGrid, and task and dependencies in workflows tools. Thus, every application functionality should be 'formatted' in this way, which might be impractical and time-consuming.

2.5. Execution environment

Gridifiers, such as CAMELotGrid, Sync Generator, and most workflow tools, generate parallel codes targeting a specific middleware. This dependency between the application and the middleware may be problematic, for instance, if the middleware project is cancelled or the middleware evolves, thus affecting the functionality of the gridifiers. For example, at the moment, the Grid community adopted service-oriented architecture, and particularly Web services; they were still under development. Furthermore, when robust and stable Grid infrastructures based on Web services were developed, both Grid infrastructures and related applications had to be consequently adapted [37].

On the other hand, GEMLCA, JGRIM, EasyFJP, and BYG provide some extension and plugging mechanisms, which break the application middleware dependency. For example, EasyFJP includes an ExecutorManager API that defines middleware-level abstractions, which particularly employs the well-known builder pattern [10] to instantiate proxies to specific middlewares.

2.6. Discussion

Table I summarizes the gridifiers mentioned throughout the previous sections.

It is worth noting that the analyzed gridifiers do not cover all the requirements of agricultural simulation applications, so they are not appropriate to gridify this kind of applications. The main drawback observed is the application anatomy prescribed by each tool. In that sense, it is not appropriate to model an agricultural simulation application with their task–entity relationships in matrix

DOI: 10.1002/cpe

Table I. Summary of gridifiers.

Work		Anatomy	Granularity	Intrusiveness	Restrictions	Middleware
GEMLCA [32]		Independent	Heavy weight	No	None	Independent
Sync Generator [30]		Divide and conquer	Light weight	Yes	Code conventions	Satin [38]
JGRIM [9]		Component oriented	Medium weight	No	Code conventions	Independent
EasyFJP [10]		Divide and conquer	Light weight	No	Code conventions	Independent
BYG [31]		Bag of tasks and divide and conquer	Medium weight and Light weight	No	Code conventions	Independent
CAMELotGrid [18]		Cellular automata	Light weight	No	Unique anatomy	Globus [39]
Workflow tools [40]	Askalon	Data flow	Light weight	No	Unique anatomy	Specific middleware, at the exception of g-Eclipse Workflow Builder
	Taverna, Pegasus, and Kepler	Workflow and data flow	Medium weight and fine weight	Yes		
	The rest of the workflow tools	Workflow	Medium weight	No		
ASAG		Workflow and data flow	Medium weight and Light weight	No	Code conventions	Independent

ASAG, Agricultural Simulation Applications Gridifier.

arrays of states cells related by a mathematical function as CAMELotGrid suggests. The complexity of developing, testing, and maintaining a whole-farm system represented in CA format could be unmanageable. Then Sync Generator, BYG, and EasyFJP, which exploit the divide and conquer anatomy, are built around the recursion concept, which cannot be elegantly applied to process dependent tasks that iteratively modifies an entity set. On the other hand, heavy-weight tools, which wrap all application binary code with no anatomy restriction (e.g., GEMLCA), and medium-weight tools, such as tools following the component-oriented anatomy (e.g., JGRIM) and workflows that do not support the data flow anatomy, prevent the application from taking advantage of light-weight parallelization opportunities offered by agricultural simulation applications, as it will be seen in the next section.

Besides, workflows that support the data-flow-only anatomy, such as Askalon, or workflows that are combined with the data flow anatomy, such as Taverna, Pegasus, and Kepler, are not focused in the relationship task–entity presented by agricultural simulation applications [13, 14, 41–44]. In these applications, entities are first-class objects, which means that they can be passed as a parameter, returned from a subroutine, or assigned into a variable *as is* [45]. These entities are composed by attributes that represent the entity status, and an attribute could be associated to a primitive type – such as integer or float – or another entity or list of entities. So, tasks change the entities status by updating their attributes, and the same set of entities flows from task to task. Contrarily, workflow tools load input data from files storing lightly structured data (e.g., flat records, byte/number streams, and tabular data); thus, they could not take advantage of the hierarchical entity structure to improve parallelization and distribution via entity-centric speeding up techniques. For example, an entity, contrary to a file or pieces of it, could be easily made identifiable all over a Grid because it represents a particular biophysical element of a particular farm, so it could be stored/replicated by using distributed key-value databases and processed in any node of the Grid.

DOI: 10.1002/cpe

Moreover, these workflows aim to enable rapid analysis of large amounts of data for scientific applications. For example, in the work of Altintas *et al*. [46], bioKepler was created to facilitate the development of Kepler workflows for integrated analysis of large DNA sequence dataset. Moreover, in the work of Qin *et al*. [47], the language for describing applications of the Askalon workflow tool was extended to support large heterogeneous datasets. Hence, Grid workflows that support the data flow anatomy are in general focused on large-scale data processing. However, agricultural simulation applications handle a moderate amount of data in the form of entities that represent the biophysical system given by the farm. Lastly, workflow tools aim at scientific environments where scientists need to model their experiments in a simple and fast way. However, the availability of workflow tools in this context has produced a lack of standardization and simplicity in the sense that scientists who are neither software developers nor scripting language experts use these tools [48].

It is worth to notice that an agricultural simulation application could be gridified by using workflow tools as well as CA or the tools that exploit the divide and conquer anatomy, but the design of the application would be forced to be compliant to the anatomy expected by these tools. Additionally, many agricultural simulation applications are already built, so the re-factoring process could be very expensive. In the area of gridification technologies, it is already known that reengineering sequential codes prior to obtaining their parallel counterpart is effort demanding [3], and thus, gridification approaches based on compilation unit modification only – that is, the one followed in this work – are preferred [3]. An example evidencing this situation – which happened before the proposed method by this paper was completed – in the studied domain is APSIM, which was manually gridified without re-factoring it to these gridification tools [15]. This tool was gridified to answer a particular domain question about wheat growth in certain areas of Australia, and the parallelization and distribution was based in climate-soil subareas that are homogeneous. So, if the question or the studied subareas change, the parallelization and distribution strategy may be outdated. In that sense, ASAG, which is based on the entity–task anatomy to which APSIM adheres [13], could be used while providing more flexibility.

Given these facts, ASAG is proposed to support the gridification of applications by exploiting the workflow–data flow anatomy combination for cleanly modeling task–entity interactions, as it will be seen in the next section. However, unlike the analyzed tools, our method simultaneously promotes other known benefits of gridifiers [3]; namely, medium-weight and light-weight gridification granularities, low code intrusiveness, and middleware independence.

3. GRIDIFICATION APPROACH

As mentioned earlier, many agricultural simulation applications written in object-oriented languages often consist of two types of well-defined elements. The first type of element represents the entities that model the biophysical system, and the second type of element represents the actions, also called tasks, that modify the contents of these entities. This two-element organization can be found for example in the models proposed by Machado *et al*. [14], by Romera *et al*. [41], by Good *et al*. [42], and by Keating *et al*. [13], as well as the frameworks proposed by Sherlock *et al*. [43] and Hillyer *et al*. [44]. Additionally, Jones *et al*. [17] described how an agricultural simulation model should be designed, suggesting the use of this organization. Overall, the common behavior of these models lies in the interaction between entities and tasks. Depending on its functionality, each task modifies properties from a subset of the biophysical entities, which may in turn activate more tasks. Thus, the interaction between tasks is implicit through the modification of shared entities.

Figure 1 summarizes the model and behavior mentioned earlier, where a biophysical entity has properties representing an element of a real biophysical system, such as an ANIMAL. For example, the entity ANIMAL has properties such as *live weight*, *live weight gain, pasture intake*, and *supplement intake*. Moreover, entities can be grouped in container entities, where the entity ANIMAL can be grouped in a container entity called HERD. The entity HERD has properties such as *pasture allocation* and *feed supplementation*, and all instances of HERD could be in turn grouped in an entity FARM. In this way, a complete biophysical system such as a farm may be designed starting with a single 'father' entity that is composed of smaller (container or simple) entities.

DOI: 10.1002/cpe

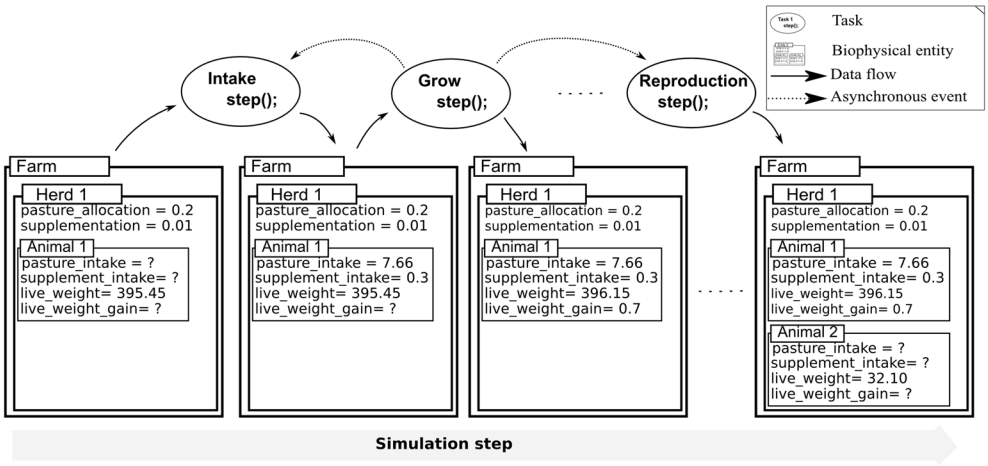


Figure 1. Agricultural simulation applications: overview.

On the other hand, a task modifies entities depending on its respective purpose (Figure 1). There are two types of tasks: namely, the tasks that are expressed as a mathematical model that updates entity properties or *property update tasks*, and the ones that move, add, or delete sub-entities to/from a container entity, or *rule-driven tasks*. Examples of the first type of tasks are the Intake and Grow tasks that act on entities ANIMAL updating the properties *pasture intake* and *supplement intake*, and *live weight* and *live weight gain*, respectively (Figure 1). Likewise, an animal Reproduction task on an entity HERD that creates a new entity ANIMAL (new born) is an example of the second type of task (Figure 1).

Property update tasks need entities and their properties as input to update properties of the same or another entity. When this kind of tasks is followed by another property update task that modifies the same entity type, a data dependence occurs. Therefore, after the modification of an entity by a property update task, the entity is released and now can be modified by the follower property update task. In short, a data dependency between such tasks may exist because tasks need information produced by other tasks. An example of task dependency is represented by a task that estimates intakes of entities ANIMAL and another task that grows entities ANIMAL (Figure 1). The intake animals task updates the properties *pasture intake* and *supplement intake,* and the grow animal task uses these property computed values to calculate the new value of the properties *live weight gain* and *live weight*.

Rule-driven tasks move, add, or delete entities and may also modify properties of different types of entities. In this context, 'rule-driven' means that after performing modifications, some conditions may produce that another rule-driven or properties update task that starts its execution. For example, a calf birth during the execution of a Reproduction task may imply that the amount of pasture in the current animals paddock is not enough for feeding the herd, so a move animals task that changes the PADDOCK where the HERD is eating or a re-assignation pasture task that updates the property *pasture allocation* of the HERD should be executed.

Figure 1 also shows that these task–entity interactions take place during a *simulation step*. A simulation step is defined as a fixed time unit, such as second, minute, hour, day, month, or year. As can be seen, a simulation is composed of at least one step.

As a consequence, during a simulation step, entities and property update tasks generate a data flow execution model (Figure 1, from left to right), while rule-driven tasks generate a task flow execution model with an added implicit execution flow because of the conditional calls between tasks, which are asynchronous events that can be caught by another task to react to special simulation conditions.

From the point of view of this gridification approach, property update tasks are the most important ones; as it was suggested earlier, this kind of tasks acts on sub-entities of a father entity, thus modifying their property values via mathematical models. These models, although not extremely costly from a computational perspective, are executed by a huge number of times, which make

DOI: 10.1002/cpe

them costly in the end. From a programming perspective, property update tasks often present the following code structure, which is illustrated in the context of the tool discussed in the following section:

```
class TaskN{
...
    public void step(Entity e){
        for(SubEntity eSub : e.getSubEntities()){
            propertiesUpdate(eSub);
        }
    }
...
}
```

This task code structure generates data flow dependencies at different entity levels, which means that when a task finishes processing an entity, this entity can be processed by another dependent task. The tasks that feed and grow entities ANIMAL, such as the ones in the example earlier, have the code structure presented, where the feed task updates intake-related properties, while the grow task updates weight-related properties. This data dependency, called data flow dependency, is present when input variables – that is, entity properties – of the mathematical model of a task are output variables of the mathematical model of another task.

Data-flow-like task dependencies are applied by different parallel strategies [49–51]. These parallel strategies start the execution of two communicating tasks together, and the dependent task will block until the nondependent task releases or finishes modifying the properties of a particular entity. Figure 2 shows that $Grow$ is modifying entity $Animal_1$ (right side of the figure), but that entity had firstly been modified by $Intake$ (left side of the Figure), which had released it, and now it is modifying entity $Animal_2$. Thus, the two tasks are running concurrently using a producer–consumer data scheme. In this sense, our gridification method exploits an ad hoc data flow parallel strategy between property update tasks, as will be explained in the next section.

However, a data flow strategy is not enough to represent an agricultural simulation application. In fact, rule-driven tasks, which may produce the execution of other tasks and change entity structures by adding, deleting, and moving entities from one container entity to another, do not present the same code structure as property update tasks. This kind of tasks may act in a conditional manner and does not rely on repeatedly executing mathematical models. From a programming perspective, rule-driven tasks share the *step()* method, but the content of the method vary from rule-driven task to rule-driven task. The next pseudocode illustrates an example of a rule-driven task, where in the first part of the *step()* method, the calculation is carried out, and in the second part, if some condition is met, a new event is thrown:

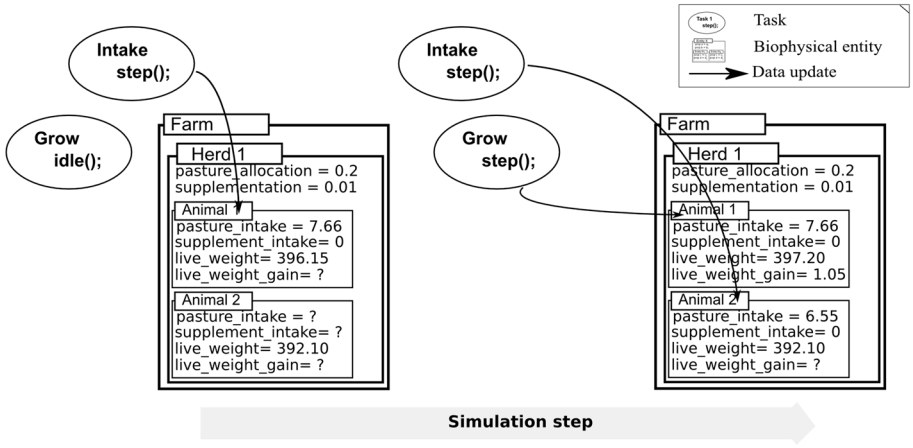


Figure 2. Agricultural simulation applications: *Grow* depends on *Intake*.

DOI: 10.1002/cpe

```
class TaskX{
...
    public void step(Simulation s){
            //calculation part
            doSomething(s);

            //event part
            if(someCondition){
                Event evt = new SomeEvent(this, s);
                EventManager.fireEvent(evt);
            }

    }
...
}
```

All in all, our new gridification approach supports the combination of task flow and data flow mechanisms [52]. Besides, the approach benefits from dependent tasks with data flow capabilities and takes advantage of performance improvement opportunities with a task flow parallelization strategy.

In addition, inserting parallel code to an agricultural simulation application should be implicit from the programmer's perspective. Thus, developers should focus on writing the business logic code of their agricultural application regardless of the parallel execution sentences. Then, by using the tool that materializes the ASAG gridification approach, the programming language sentences related to parallel execution are included automatically during the compilation stage of the simulation application. Conversely, when relying on explicit Grid programming, developers manipulate methods offered by middleware APIs to orchestrate parallel computations. This alternative can generate better execution times, but developers with expertise in parallel computers are strongly recommended to perform gridification [3].

In short, ASAG exploits agricultural simulation application anatomy to enhance the overall execution performance, minimizing the insertion of parallel code into dependent tasks to maintain code readability. In addition, this approach provides users with what follows:

- Middleware independence: Grid middlewares that support master–worker parallelism can be easily interchanged.
- Low expertise: Developers with high expertise in parallel programming are not required. This achievement is fundamental because of the agricultural application domain, where most of the work group members lack high expertise in parallel programming. However, the developer may modify the generated code with explicit parallelism in case an improvement is required [53].

In summary, ASAG offers agricultural simulation applications represented by the structure described earlier the opportunity to be gridified. From existing gridifiers, ASAG combines the *fork* and *join* concept [4] (many instances of a task are allowed to execute with concurrency managed automatically), low intrusiveness characteristics, and middleware abstraction. Again, the contribution of this approach is the gridification of these applications that are modeled as task–entity interactions by combining task flow–data flow parallelization strategies.

4. JASAG

The JASAG is the tool that implements the ASAG gridification method explained previously for Java middlewares and applications. The selection of Java as a first implementation of the method is due to the popularity gained by this language because of its 'write once, run anywhere' property that promotes platform independence, which is very useful for implementing distributed platforms, and the fact that its delivered performance is competitive with respect to that of conventional High Performance Computing (HPC) languages [54]. Additionally, the agricultural simulator used in the experiments described in Section 5, that is, Simugan [14], is also implemented in Java.

The JASAG supports the ASAG gridification method by providing two separate but related software elements (Figure 3). The first one, called Gridificator, is responsible for helping users to

DOI: 10.1002/cpe

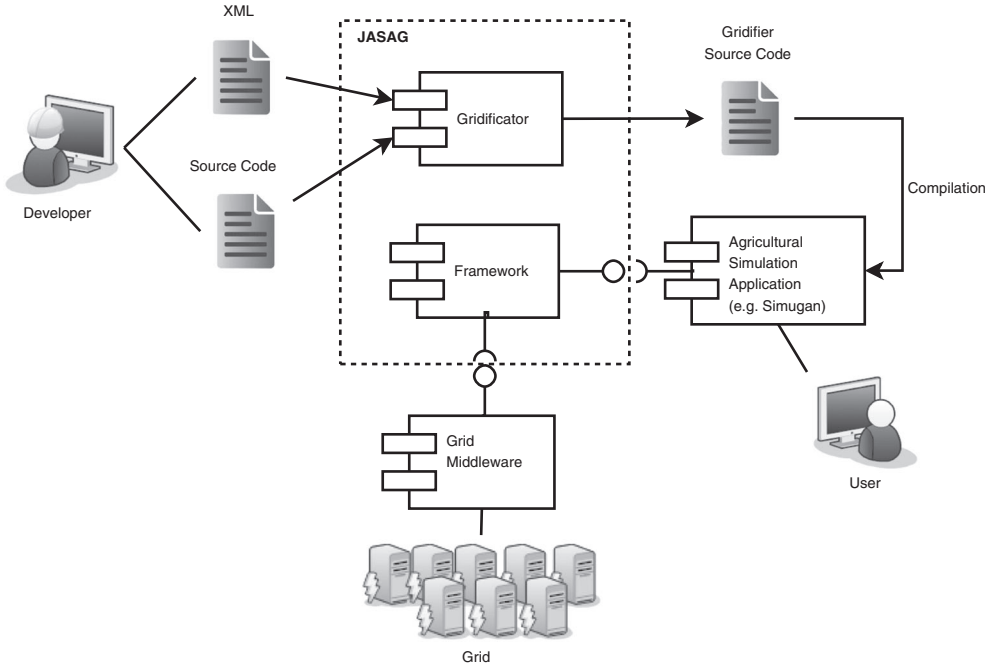


Figure 3. JASAG: overview.

turn sequential application codes into a parallel, distributed application codes. The second element, called Framework, is a framework that provides parallel and distributed execution support for the agricultural simulation application obtained from the compilation of gridified codes produced by the Gridificator.

4.1. The Gridificator

The Gridificator is implemented as an Eclipse plug-in. This plug-in helps users to indicate in their own code the biophysical entities, the tasks, and the relationships between them. Based on this information, the plug-in produces parallel code exploiting task flow and data flow dependencies. Thus, the plug-in divides its activities into two stages. The first stage is where the users identified in their own sequential source code biophysical entities, tasks, and task dependencies. The second stage is where the sequential code transformation is carried out and the Grid-enabled code that uses the functionality implemented in the Framework element is obtained.

In the first stage, by using the plug-in graphical interface, a user identifies entities, tasks, and dependencies in his or her source code, which are compiled by the plug-in into an XML file such as what follows:

```
<configuration>
    <!-- biophysical entities -->
    <entity name="simulation" class="path.to.class.Simulation"/>
    <entity name="farm" class="path.to.class.Farm"/>
    <entity name="herd" class="path.to.class.Herd"/>
    <entity name="animal" class="path.to.class.Animal"/>
    ...
    <!-- tasks -->
    <task name="growPasture" class="path.to.class.Reproduction"/><!--
        rule-driven task -->
    <task name="adjustAnimalAllowance" class="path.to.class.Reproduction
        "/><!-- rule-driven task -->
    <task name="intake" class="path.to.class.Intake"/><!-- property
        update task -->
    <task name="grow" class="path.to.class.Grow"/><!-- property update
        task -->
```


DOI: 10.1002/cpe

```
13      <task name="reproduction" class="path.to.class.Reproduction"/><!--
            rule-driven task -->
14      <task name="pregnancyTest" class="path.to.class.PregnancyTest"/><!--
             rule-driven task -->
15      ...
16      <dependencies>
17        <!-- task flow dependency -->
18        <dependency source="growPasture" destination="intake"/>
19        <!-- task flow dependency -->
20        <dependency source="adjustAnimalAllowance" destination="intake"/>
21        <!-- data flow dependency -->
22        <dependency source="intake" destination="grow">
23          <entity name=animal/>
24        </dependency>
25        <!-- task flow dependency -->
26        <dependency source="grow" destination="reproduction"/>
27        <!-- task flow dependency -->
28        <dependency source="reproduction" destination="pregnancyTest"/>
29      </dependencies>
30  ...
31  </configuration>
```

The XML shows the entity-related tags (lines 2 to 6), the task-related tags (lines 8 to 14), and the dependency-related tags (lines 16 to 29). The attributes of entity and task tags are the name of the element tag and the path to the class that implements that entity or task in the source code. The dependency tag is composed of the source and destination attributes, which point to a name of a task tag defined elsewhere in the file. If the dependency tag does not contain an entity tag, it is a task flow dependency. Otherwise, it is a data flow dependency. The dependency defined in lines 21 to 24 was shown in the example of Section 3, which involves the property update tasks Intake and Grow and its data dependency. All task classes have a *step()* method, which is where the main difference between property update tasks and rule-driven tasks source codes lies. The inner structure of this method varies from rule-driven task to rule-driven task, but property update tasks have a similar 'for' statement that cycles entities to access their properties or decomposing them (father entities). The source code of the property update task Intake before applying the gridification process would look like the following:

```
1   public class Intake {
2       public Intake() {}
3
4       public void step(Simulation simulation) {
5           for(Animal animal : simulation.getFarm().getAnimals()){
6               animalUpdate(animal);
7           }
8       }
9   }
```

In case of the property update task Grow, the source code is almost the same, but the difference lies in the mathematical algorithm implemented in the *animalUpdate(animal)* method (line 6).

Then, in the second stage with the sequential source code and the XML file generated in the first stage as input, the gridifier modifies the task source code depending on the framework components where it would be hooked. Thus, after the gridification process, the Intake task source code would be as follows (the Grow task would be modified in a similar way):

```
1   public class Intake implements Task{
2       public Intake() {}
3
4       Intermediary intermediary;
5       //getter and setter for intermediary
6
7       public void step(Simulation simulation) {
8           for(Animal animal : simulation.getFarm().getAnimals()){
9               SubTask<Animal> subtask = new SubTask<>(animal.getID()){
10                  public void processEntity(Animal animal){
11                      animalUpdate(animal);
```


DOI: 10.1002/cpe

```
12                  });
13              intermediary.execute(subtask);
14          }
15      }
16
17      @Override
18      public void run(){
19          this.step(simulation);
20      }
21  }
```

As illustrated, both the Intake task and the Grow task implement the interface task and have an Intermediary attribute at line 4, which are part of the Framework (Section 4.2), in order to execute tasks in a parallel and distributed way. Now, inside the *step()* method (line 9), an instance of the *SubTask* class is created, and the implementation of its abstract method *processEntity(entity)* is the sequential *animalUpdate(animal)* method. Then, at line 13, the Intermediary is called with this subtask as a parameter. These lines involve an interaction with the library that enables data flow distributed/parallel execution. Finally, a *run()* method is added at line 18 because the task interface implements the *java.lang.Runnable interfac*e. This *run* method calls the *step* method inside its body, which enables task flow distributed/parallel execution. Thus, in rule-driven tasks, the only modifications needed are implementing the interface task and including the *run* method.

To recognize, add, or replace the statements in the input source code, the Eclipse JDT Core library [55] is used. This library enables the Gridificator to access the abstract syntax tree of a class and modify its content. The next pseudocode outlines the recognition and replacement of the 'for' statement inside the *step()* method:

```
1   public class GridVisitor extends org.eclipse.jdt.core.ASTVisitor {
2
3       public boolean visit(ForStatement node) {
4           //applies modifiers for 'for' statements
5           List<ModifierByCriteria> listOfModifier = hashmap.get(node.
                getType());
6           for(ModifierByCriteria mbc:listOfModifier){
7               mbc.applyModifier(node)
8           }
9           return true;
10      }
11
12      public boolean visit(TypeDeclaration node) {
13          ....
14      }
15
16      public boolean visit(ExpressionStatement node) {
17          ....
18      }
19  }
20
21  public class ModifierByCriteria {
22
23      GridCriterion gc;
24      GridModifier gm;
25
26      //If the criterion is approved, the modification is realized
27      public void applyModifier(ASTNode node) {
28          if(gc.verifyCriterion(node){
29              gm.apply(node);
30          }
31      }
32
33  }
34
35  public class ForCriterion implements GridCriterion{
36      String className;
37      String method;
38      String entityName;
39
40      public boolean verifyCriterion(ASTNode node){
```


DOI: 10.1002/cpe

```
41          //checks if it is a 'for' statement of a task class, inside the
                'step()' method, for the requeried entity
42          if(isEntityFor(node, className, method, entityName)) {
43              return true;
44          }
45          return false;
46      }
47  }
48
49  public class ForModifier implements GridModifier{
50
51      public apply(ASTNode node){
52          //Creates the subtask as String
53          String subTaskCode = getSubTaskCode(node);
54          //Turns the String in statements
55          Statement newBody = addaptSubtask(node, subTaskCode);
56          //Sets the new statements to the boby
57          setNewBody(node, newBody);
58      }
59
60  }
```

The Gridificator main interfaces are the GridCriterion and GridModifier. Each interface implementation represents, in case of GridCriterion, something to find in the original source code (criterion), and, in case of GridModifier, how it is changed (modification). By visiting the complete source code structure, if a criterion is found, a modification is applied. Meanwhile, the class ModifierByCriteria has a concrete instance of a criterion and a modification; thus, its functionality is to apply the modification if the criterion is satisfied. This implementation uses the strategy design pattern [56], so if the Gridificator software evolution needs new pieces of code that should be identified and modified, this is the place where they should be added.

In addition to tasks gridification, biophysical entities also have to be rewritten by the Gridificator. The entities have to be potentially shared by every node in the Grid, so every node executing tasks could have access to them. Thus, in order to Grid-enable entities, a universally unique identifier (UUID) and shared distributed memory access properties are added. Additionally, references to entity properties in sequential entity classes must be replaced by a UUID, and local access to those properties must be carried out through their getter methods, a code convention frequently used in Java development [57, 58]. The next code exemplifies a sequential entity:

```
1   public class Mob{
2       //attributes, plus getters & setters
3
4       // (Father) Entity that contains this entity
5       private Farm farm;
6
7       public Farm getFarm(){
8           return farm;
9       }
10
11      //Example method
12      public void doSomething(){
13          double area = farm.getArea();
14          ...
15      }
16  }
```

After applying the gridification process, by modifying class attributes and methods as explained, the next code exemplifies a Grid-enabled entity:

```
1
2   public class Mob implements EntityMemory{
3       //attributes plus getters & setters
4
5       private UUID Id;// new attribute: shared memory access
6
7       private SharedMemory sharedMemory; //new attribute: shared memory
            access
```


DOI: 10.1002/cpe

```
8       //getters & setters for id, sharedMemory
9
10      //UUID of the father Entity that contains this entity
11      private UUID farmUUID; //type changed and name changed
12
13      public Farm getFarm(){
14          return sharedMemory.get(farm); //changed to share memory access
15      }
16
17      //Modified example method
18      public void doSomething(){
19          double area = this.getFarm().getArea();//changed to ensure
                access to shared memory
20          ...
21      }
22
23  }
```

All in all, the gridification approach automatically inserts parallel-related code, thus reducing developers involvement. In addition, to reduce the need for explicit parallel programming, the dependency injection (DI) pattern [35] was selected to separate Grid-aware code from business logic code over annotations in code [59, 60] and metaobjects usage [61] because of the higher decoupling that DI provides. In this sense, the attributes of types Intermediary and SharedMemory used in tasks and entities respectively are Java interfaces; thus, it is up to the DI container to create and set (hence 'inject') an object instance implementing the interfaces. The concrete implementations of Intermediary and SharedMemory interfaces enable the application to execute simulation tasks in parallel and distributed. The DI container and the concrete implementation of these interfaces are provided by the Framework (Section 4.2). As a consequence, the combination of DI with Java Beans leads to a separation between the application logic code and the parallel-related code.

4.2. The Framework

The parallel task executor framework (Figure 4) is in charge of executing gridified codes produced by the Gridificator. The framework, implemented in Java, consists of four main software components: *Simulator*, *Simulation*, *Intermediary*, and *Scheduler*. The *Simulator* component administrates every new simulation instance started by the user. *Simulator* uses the component *Intermediary*, which is the interface variable added by the Gridificator (Section 4.1), and acts as a proxy between a particular middleware – such as GridGain and Ibis – where the execution takes place and the *Scheduler* component that, through the Intermediary, launches each task to execute in parallel. Finally, the gridified application code, which consists of Grid-aware biophysical entities and tasks, is injected in the Simulation component because the EntityMemory and task interfaces, added by the Gridificator, are essential parts of the simulation abstraction represented by this component.

The Scheduler component creates a direct acyclic task graph based on the XML defined by the user. This component uses a breadth-first search trace path approach for managing task execution

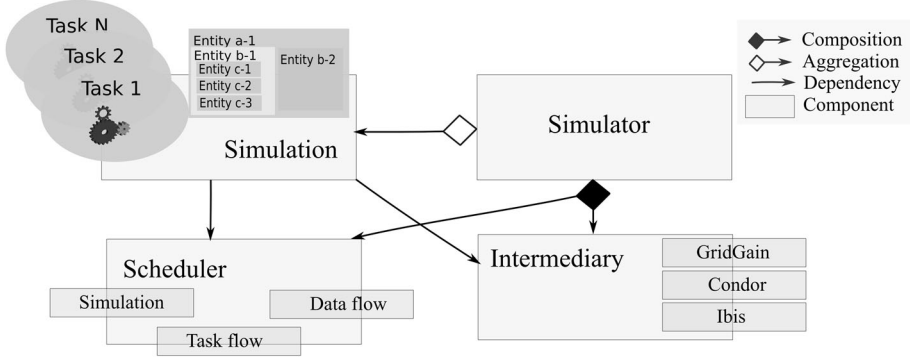


Figure 4. Parallel tasks execution framework: main components.

DOI: 10.1002/cpe

strategies at runtime. The possible strategies are *simulation*, *task flow*, and *data flow*. The simulation strategy runs each simulation instance in parallel, but their tasks run sequentially. The task flow strategy adds to the simulation strategy the possibility of running in parallel nondependent tasks and tasks activated because of rule-driven task execution. Every task launched because of a condition fulfilled for a rule-driven task; it is executed in parallel, but all these tasks must be finished before the next task in the graph is executed. For example, taking into account the XML file defined in Section 4.1, when multiple births happen at a particular herd during the execution of the Reproduction task, this may produce that pasture available in the paddock where the herd is eating is not enough. For that reason, the MoveHerd task could be executed in parallel, while the Reproduction task continues its execution. But, the PregnancyTest task (the next task in the graph according to the XML file) must wait until both tasks finish. Finally, the data flow strategy extends the task flow strategy with the possibility of running dependent tasks using the parallel strategy under the consumer–producer entity scheme explained in Section 3. In other words, when a task finishes processing an instance of an entity, the instance is released, and then, the dependent task can lock it in order to perform its own processing.

The Scheduler, through the Intermediary, can run tasks in any node of the Grid, and for that reason, entities must be available from every node. Thus, together with the task execution middleware abstraction, the *Intermediary* manages distributed memory accesses. As a consequence, the whole entity hierarchy is stored in a distributed key-value database partially mapped to main memory in nodes to increase access speed (Figure 5). Parallel tasks require access to any stored entity. If a task acquires the lock of an entity, any other task requiring that entity has to wait for the associated lock. The next code shows the abstract class SubTask contained in the Simulation component and inserted by the Gridificator, which illustrates the behavior described:

```
1  public abstract class SubTask<E extends EntityMemory> implements
       Runnable {
2      UUID entityUUID;
3
4      public void run () {
5          //wait until the task gets the lock over the entity
6          sharedMemory.lock(entityUUID);
7          //get the entity from the distributed database
8          E entity = sharedMemory.get(entityUUID);
9          //process the entity
10         processEntity(entity);
11         //save the entity
12         sharedMemory.put(entityUUID , entity);
13         //release the entity
14         sharedMemory.unlock(entityUUID);
15     }
16
17     public abstract void processEntity(E entity);
18 }
```

The selection of Not Only SQL (NoSQL) databases – such as the selected key-value database – over relational databases is based on the performance delivered by the former for simple operations such as reads and writes that involve small amounts of data [62]. In that sense, agricultural

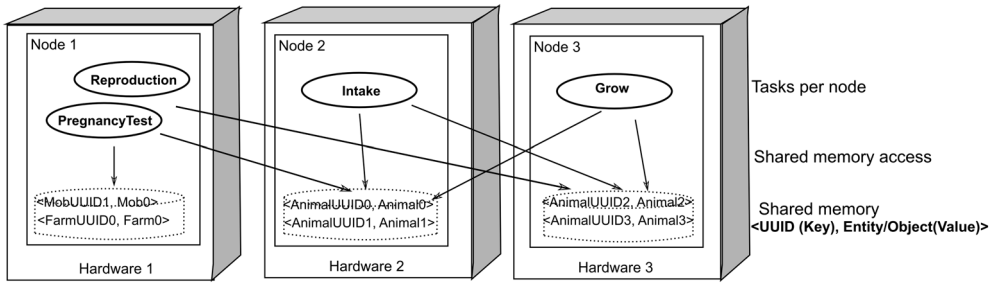


Figure 5. Shared memory access.

DOI: 10.1002/cpe

simulation applications are composed of tasks that update Java attributes of entities many times during a simulation. For example, property update tasks Intake and Grow read from the database entity attributes needed by their mathematical models, and after the calculations, the tasks write to the results to the database, which in turn correspond to another entity attributes. Thus, a key-value database is a good choice. In addition to performance, key-value databases promote scalability, which is a desired nonfunctional requirement in distributed environments.

Regarding the particular NoSQL database employed to deploy JASAG, any database could be used, but the way it is integrated into the framework depends on the resource allocator being used. In this sense, two possible scenarios arise: either the resource allocator comes with a built-in NoSQL database or supports seamless integration with third-party NoSQL databases (e.g., GridGain), or neither of these hold (e.g., Ibis and Condor). In this latter case, both an NoSQL database and a proper client API to access the database should be installed on machines. For the purposes of this paper, we have used JASAG in conjunction with the GridGain middleware, which provides a built-in NoSQL database called in-memory data grid (IMDG)[‡] and was configured to use round-robin load balancing for resource allocation.

Particularly, IMDG is essentially a distributed, in-RAM key-value cache backed up with disk-based permanent storage. To modify keys and values, the cache is equipped with a simple two-phase commit protocol over an ad hoc socket-based network communication protocol. Lastly, keys and values can be transferred through the network in three formats: binary (serialized Java objects), XML, and JavaScript Object Notation (JSON).

5. EVALUATION AND DISCUSSION

In order to evaluate the proposed gridification method and the tool that implements it, Simugan [14] was used as a case study. Simugan is a whole-farm simulator, oriented to assist research, teaching, and technology transfer of alternative beef cattle production systems. A simulation is defined within a *scenario*, which contains entities, initial values, and conditional rules that are applied to tasks in order to represent decision making in a particular farm setting. Users build a scenario by accessing a Web site where, through different user interfaces, they create, save, modify, retrieve, or delete their own scenario(s). Simulation outcomes are downloaded as a spreadsheet file.

Simugan is mainly used in agricultural undergraduate and graduate courses in Argentina to explore by proof and error strategies best action courses for alternative simulated farms. Furthermore, different graduate theses make an intensive use of that simulation tool for research purposes. At present, the expanded use of Simugan in additional agricultural schools across Argentina is under promotion, so a huge simulation burden will be produced sooner than later. Present and future uses of Simugan highlight the need to run these simulations in parallel. Additional to scholarly usage, Simugan is used by Agricultural Technology National Institute (INTA, Argentina) researchers to conduct their investigations. INTA is a state agency in charge of technology innovation and research in agricultural topics. An example of Simugan usage by an INTA researcher is from the work of Berger *et al*. [63], who studies the maize silage and oat winter forage crop impact on cow-calf systems. In our view, this performance requirements justify the application of the gridification method to Simugan.

The comparison was circumscribed to using the ASAG gridification method and its Java implementation because the software refactorings needed to adapt Simugan to workflow tools, CA, or the tools that exploit the divide and conquer anatomy could have been very time-consuming and impractical, as explained in Section 2.6. But even more important, the anatomy supported by the proposed gridification method matches the entity–task anatomy present in agricultural simulation applications. Nevertheless, the pure workflow anatomy is represented by the task flow parallelization strategy of ASAG. Then, at least from a conceptual (not technical) perspective, the hybrid parallelization strategy of ASAG is compared against the pure (task-based) parallelization strategy followed by most workflow tools.

[‡]http://gridgain.com/developer-central/in-memory-data-fabric/in-memory-data-grid/.

DOI: 10.1002/cpe

Based on what has been said, gridifying Simugan first involved the creation of an XML file needed by the Gridificator (XML defined in Section 4.1). Twenty one entities, such as FARM, ANIMAL, CALF, HERD, and PADDOCK, 32 tasks such as GrowPasture, ControlGrazingPaddocks, UpdatePastureCover, AdjustPastureComposition, MoveHerds, FeedIntake, and GrowAnimals, and the dependencies between tasks were included in that file. The 32 tasks generate a 32-node acyclic graph; thus, during a simulation step, 32 sub-steps are needed to execute every task sequentially (Figure 6a). However, if the task flow and data flow dependencies between tasks are exploited, all tasks are executed in 28 (Figure 6b) and 21 (Figure 6c) sub-steps, respectively.

After gridification, the gridified code was run in a cluster with 42 cores distributed in seven nodes, with the characteristics shown in Table II. All nodes run the GridGain 5.2 [60] middleware (GG from now on). GG is a mature Java middleware that provides a robust solution to both distributed storage and task execution. For distributed storage, the GG IMDG was used, which is an object-based, atomicity, consistency, isolation, durability transactional, in-memory key-value store. It was configured in a way that the overall dataset is divided equally between participating nodes, essentially creating one huge distributed in-memory store. On the other hand, the task execution was configured with a round-robin load balancing in order to guarantee that every node in the execution environment is equally loaded. Furthermore, fault tolerance and security – as well as other middleware-level services – are common in most classical Grid middlewares such as Satin [59, 64, 65], JPPF [66], or GG [60]. For that reason, these services are fully delegated by JASAG to the underlying middleware.

The performance metrics used for assessing Simugan simulations included execution time, speedup, efficiency, memory usage, CPU usage, and bytes transferred between nodes. These metrics were calculated at different simulation loads: that is, X *simulations* executing concurrently with $X = 1, 10, 50, 100, 150, 200$. Moreover, simulations are run with different parallelization

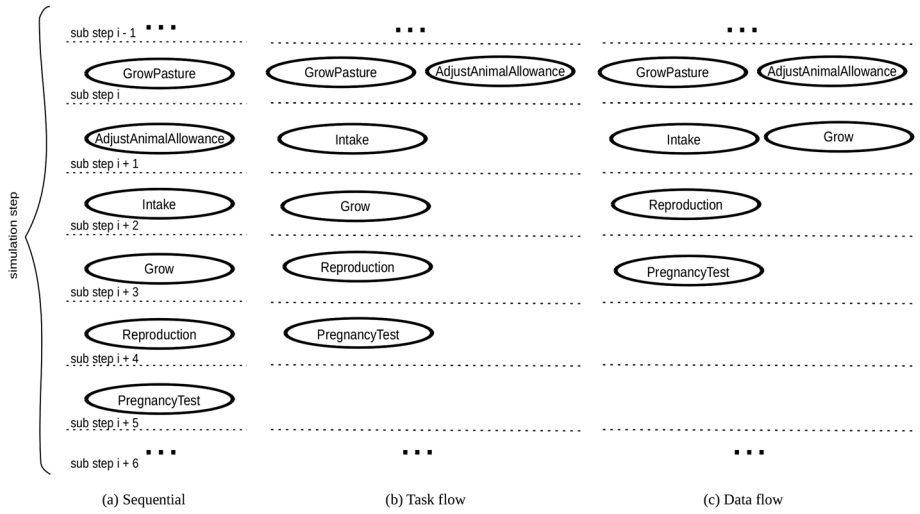


Figure 6. A portion of the tasks execution graph generated from the XML configuration file defined for Simugan.

Table II. Execution environment: node characteristics.

Node	CPU	Net controller	RAM (GB)
N1	AMD FX(tm)-6100,	Realtek Semiconductor	
N2	Six-core processor,	Co., Ltd.	8
N3	running at 3.6 GHz	RTL8111/8168B PCI	
N4		Express Gigabit	
N5	AMD Phenom(tm) II	Qualcomm Atheros	
N6	X6 1055T processor,	AR2417 Wireless	16
N7	running at 2.8 GHz	Network adapter	

DOI: 10.1002/cpe

strategies: (i) sequential, the original Simugan code run in a single node from one of the nodes with 16 GB RAM, (ii) simulation-level parallelization, where simulations are run concurrently as black boxes, (iii) task flow parallelization, where simulations and nondependent tasks inside simulations run concurrently, and (iv) data flow parallelization, where simulations, nondependent tasks inside simulations, and property update dependent tasks run concurrently.

Figure 7 shows the simulation execution time with different loads and parallelization strategies. The data flow parallelization strategy shows the lowest execution times followed by the simulation-level and the task flow parallelization strategies, and lastly the sequential execution. The task flow strategy presents worse times compared with the simulation-level parallelization strategy, as there is not a large amount of independent tasks that can run in parallel: as it was mentioned earlier, simulation-level parallelization takes 32 sub-steps, while task flow takes 28 sub-steps, so 4 tasks can run in parallel with other tasks during a simulation step. Thus, the parallelization and synchronization overheads produced by the Framework to administrate task flow execution lead to a negative impact in the overall execution time. However, the Framework implementation of the data flow strategy is based on the implementation of the task flow strategy, which, with the addition of the producer–consumer data scheme that enables concurrent execution of dependent property update tasks, produces the best execution times.

Besides execution times, Table III shows the speedup obtained by each parallelization strategy with respect to the sequential execution. The speedup metric measures how faster the parallel version of a code is versus its sequential version; that is, $S_c = \frac{T_1}{T_c}$, where c is the number of cores, T_1 is the sequential execution time on one core, and T_c is the parallel execution time using the c cores. The ideal value for this metric, when cache effect is not present (super linear speedup) as in this case, is $S_c = c$, but obtaining this value would be possible only if byte transmission and parallel execution administration would not had any overhead. Another related metrics is efficiency, which measures

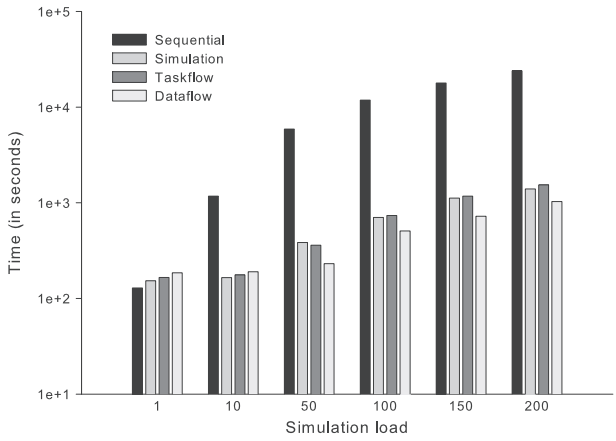
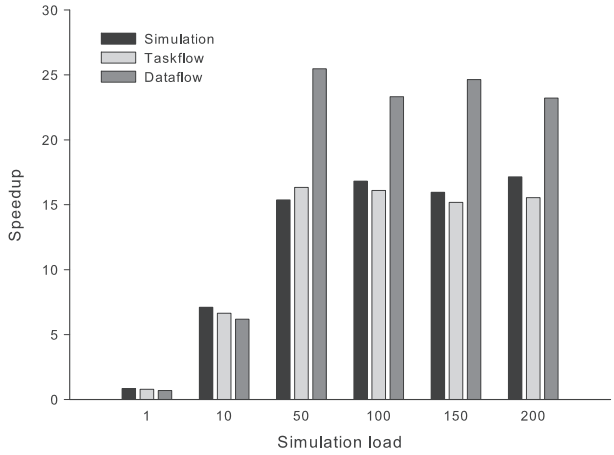


Figure 7. Execution time (logarithmic scale).

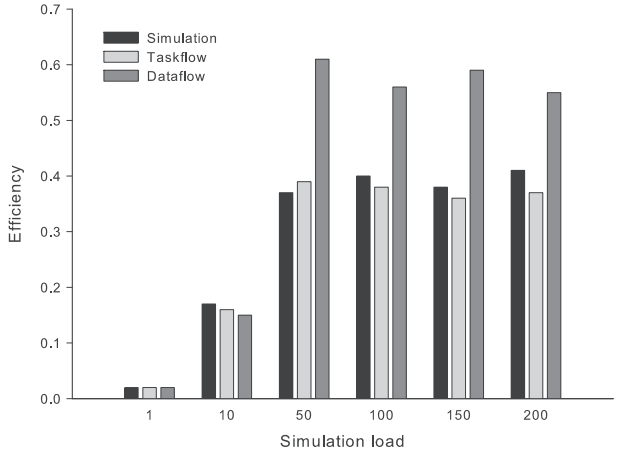
Table III. Execution time and speedups.

Simulation load	Sequential	Simulation-level parallelization	Task flow	Data flow	Speedup		
	Execution time in seconds				Simulation-level	Task flow	Data flow
1	129	153	165	185	0.84	0.78	0.69
10	1174	165	177	189	7.10	6.63	6.18
50	5895	384	361	231	15.37	16.33	25.47
100	11,837	704	735	507	16.81	16.10	23.32
150	17,854	1119	1175	724	15.96	15.19	24.64
200	23,921	1395	1538	1030	17.15	15.55	23.21

DOI: 10.1002/cpe



(a) Speedup



(b) Efficiency

Figure 8. Performance results different parallelization strategies and simulation loads.

how good core usage is in relation to the time needed to perform communication and task result synchronization: $E_c = \frac{S_c}{c} = \frac{T_1}{pT_c}$. In that sense, Figure 8 shows results for the metrics mentioned. Particularly, Figure 8a and b shows speedup and efficiency results, respectively. These metrics reach its maximum when the number of simulations is close to the number of cores, then the values remain relatively similar.

The CPU average usage metric (Figure 9) was calculated from each core usage within 1-min period, and then, these values were averaged. The way the metric was calculated was supported by the CPU usage behavior observed through the GG visor [67], allowing us to check in each experiment that CPU usage had three stages as it was supposed. In the first stage, the CPU usage increases quickly as simulations are launched. In the second stage, the CPU usage is relatively constant during simulation execution. Lastly, in the last stage, the CPU usage decreases as the simulations finish. Thus, it was important to measure the CPU average usage of the total environment by sampling CPU usage in the second stage mostly. The results of measurements during the second stage indicate that the CPU average usage increases as the simulation load grows, but when the simulation load is greater than the number of cores, the curve stabilizes. In other words, when the simulation load is lower than the number of cores, it is possible that not all the cores are used. Then, when the simulation load is greater than the number of cores, the curve growth depends on how long the cores are used, in other words, how much time the cores are processing tasks. Thus, the data flow

DOI: 10.1002/cpe

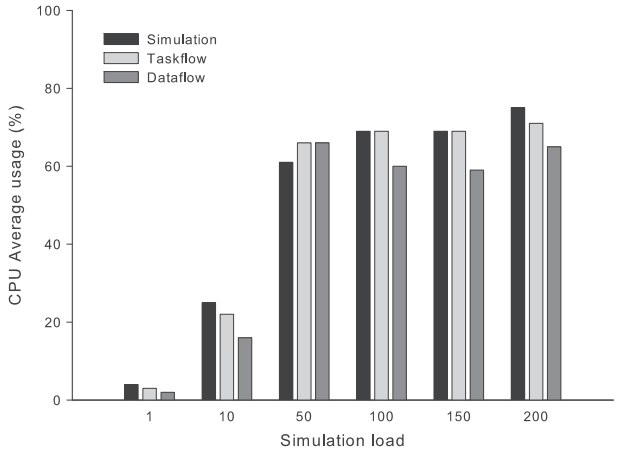


Figure 9. CPU average usage.

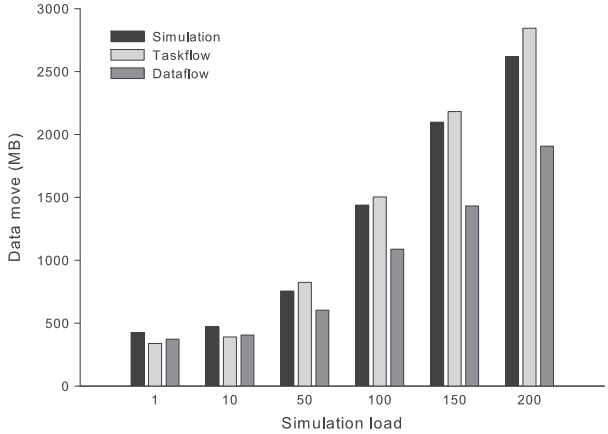


Figure 10. Total data transfer between nodes.

parallelization strategy has a lower CPU usage because its execution time to process all simulations is the lowest.

Figure 10 shows the data transfer between nodes when running each simulation load. The data flow parallelization strategy exchanges less megabytes (MB) than any other strategies. Moreover, during the execution of each simulation load, the three parallelization strategies exchange biophysical entities and control node data between nodes. The control node data includes node discovery, scheduling, fault tolerance, and authentication information, plus internal GG metrics values, among others. The difference in each simulation load between MB transferred for exchanging entities is zero because simulations have the same number of entities for the three parallelization strategies. Indeed, the difference lies in the control node data exchanged during the simulation execution time. If the MB transferred are divided by the execution time of the experiment, the difference between each parallelization strategy at different loads is negligible too. Thus, the result is explained by the data flow less execution time. In that sense, if simulation load grows, the bytes that transfer metric will grow too, because it is correlated to the execution time.

In the case of the average memory usage, similar to the average CPU usage, had three usage stages where the most important stage to measure is the second as in CPU usage metric. The second stage was calculated based on per-core memory usage values within 1-min period, and then, these values were averaged. The average memory usage increases as the simulation load increases (Figure 11), which is the expected behavior. The data flow strategy uses less average memory than the other parallelization strategies. The reasoning is the same as the one used for byte transfer. All the strategies

DOI: 10.1002/cpe

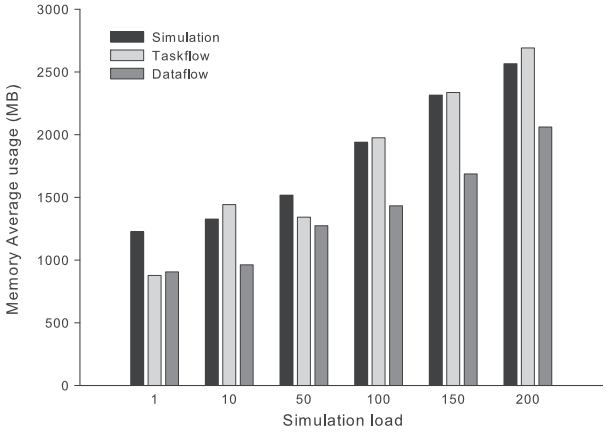


Figure 11. Average memory usage.

store the same entity bytes during the execution time, so the difference lies in the control node data along time.

Summarizing, the experiments show that parallelization strategies *simulation*, *task flow*, and *data flow* supported by the Framework enable for speedups of 17, 15, and 23 times when running 200 concurrent Simugan simulations in the experimental testbed, respectively. These improvements allow for Simugan to be used in domain-specific production and educational scenarios, because the total execution time of 200 concurrently simulations is reduced from 7 h to around 25 min (task flow parallelization strategy). Even more, the data flow parallelization strategy presents a better overall performance for Simugan.

However, the metric values may change if the dependent tasks and the number of entities change. First, if the task dependencies change, that is, more tasks can be run in parallel, the task flow might perform better than the simulation strategy. Second, if the number of entities to process in parallel by two dependent tasks is low, the data flow strategy might not present as good speedups, as in the Simugan study case. Then, it would be interesting for ASAG to exploit *policies* [10, 68]. Policies are dynamically-evaluated rules that 'throttle' task granularity, considering that the data flow allows for light-weight tasks and task flow represents medium-weight tasks. Further, policies should consider the number of entities in case a light-weight granularity is set. For example, if there is a *herd* with 10 ANIMAL entities as input for a simulation, the overhead due to light-weight parallelization might increase the data flow execution time compared with the task flow times.

6. CONCLUSION AND FUTURE WORK

As explained throughout this paper, the ASAG gridification method makes it possible for a sequential agricultural application with a specific anatomy to be turned into a parallel one. In that sense, ASAG allows developers with low expertise in gridification technologies to keep focused on the implementation and testing of the business logic code. When using the JASAG implementation, users should configure the Gridificator (Section 4.1) that does the necessary code conversions automatically. Then, the converted sequential code is run with the Framework (Section 4.2) in a Grid. The code produced by the Gridificator can be optionally modified by expert users to take advantage of middleware features.

Moreover, the Grid-enabled agricultural simulation application is independent from a particular middleware. Therefore, the middleware must support the master–worker execution paradigm and allow the execution of applications written in Java.

Experiments showed that the use of the gridification method ASAG, and its implementation JASAG, increases Simugan agricultural simulation application performance. Thanks to this, it is now possible to build larger simulations regarding amount of entities and simulation period extension or increase the number of concurrent users of the simulation tool.

DOI: 10.1002/cpe

We are extending our work in several directions. In addition to policies [10, 68], we are studying techniques from the autonomic computing area [8] in order to manage distributed and parallel execution of parallel tasks. This area improves parallel and distributed execution by self-regulating middleware-level parameters. However, to date, these techniques were applied over pure workflows systems only; thus, it would be interesting to extend these concepts to work flow and data flow parallel execution alike.

On the other hand, we plan to study JASAG in the context of a WAN network. For this purpose, JASAG should manage the network overhead introduced if a simulation is divided among many geographically dispersed clusters for the task flow (iii) and the data flow (iv) gridification alternatives. One possibility is to develop an upper level simulation scheduler, to distribute tasks and entities of a particular simulation to the same cluster.

Additionally, a challenge presented after applying the gridification method to Simugan is how to deal with simulation results scattered in every node wherein simulation tasks are executed. In order to maximize performance, tasks are distributed among as many nodes as possible, and as a consequence, partial (but heavy) simulation results are dispersed. In that sense, just the 200-simulation load experiment generates 14 GB of biological and farm management results, so the data collection to build the final spreadsheet file is very time-consuming. As a consequence, we plan to study big data [69] techniques such as MapReduce [70] to improve overall system performance.

REFERENCES

1. Foster I, Kesselman C, Tuecke S. The anatomy of the grid: enabling scalable virtual organizations. *International Journal of High Performance Computing and Applications* 2001; **15**(3):200–222.
2. Foster I, Kesselman C. *The Grid 2: Blueprint for a New Computing Infrastructure*. The Elsevier Series in Grid Computing: San Francisco, CA, USA, 2003.
3. Mateos C, Zunino A, Campo M. A survey on approaches to gridification. *Software: Practice and Experience* 2008; **38**:523–556.
4. Mateos C, Zunino A, Hirsch M, Fernández M. Enhancing the BYG gridification tool with state-of-the-art grid scheduling mechanisms and explicit tuning support. *Advances in Engineering Software* 2012; **43**(1):27–43.
5. Fahringer T, Jugravu A. JavaSymphony: a new programming paradigm to control and synchronize locality, parallelism and load balancing for parallel and distributed computing: research articles. *Concurrency and Computation: Practice and Experience* 2005; **17**(7-8):1005–1025.
6. He T, Ni J, Wang S, Knosp B. Java grid computing library (JavaGCL) – an application framework for computational grids. *The Second International Workshop on Grid and Cooperative Computing*, Shanghai, China, 2003; 21–25.
7. Laszewski GV, Gawor J, Lane P, Rehn N, Russell M. Features of Java commodity grid kit. *Concurrency and Computation: Practice and Experience* 2003; **14**(13-15):1045–1055.
8. Rahman M, Ranjan R, Buyya R, Benatallah B. A taxonomy and survey on autonomic management of applications in grid computing environments. *Concurrency and Computation: Practice and Experience* 2011; **23**(16):1990–2019.
9. Mateos C, Zunino A, Campo M. Grid-enabling applications with JGRIM. *International Journal of Grid and High Performance Computing* 2009; **1**(3):52–72.
10. Mateos C, Zunino A, Campo M. An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. *Computer Languages, Systems & Structures* 2010; **36**(3):288–315.
11. Giorgino T, Harvey M, de Fabritiis G. Distributed computing as a virtual supercomputer: tools to run and manage large-scale BOINC simulations. *Computer Physics Communications* 2010; **181**(8):1402–1409.
12. Wang X, Yan Z, Li L. A grid computing based approach for the power system dynamic security assessment. *Computers & Electrical Engineering* 2010; **36**(3):553–564.
13. Keating BA, Carberry PS, Hammer GL, Probert ME, Robertson MJ, Holzworth D, Huth NI, Hargreaves JNG, Meinke H, Hochman Z, McLean G, Verburg K, Snow V, Dimes JP, Silburn M, Wang E, Brown S, Bristowc KL, Asseng S, Chapman J, McCown RL, Freebairn DM, Smith C J. An overview of APSIM, a model designed for farming systems simulation. *European Journal of Agronomy* 2003; **18**(3-4):267–288.
14. Machado C, Morris S, Hodgson J, Arroqui M, Mangudo P. A web-based model for simulating whole-farm beef cattle systems. *Computers and Electronics in Agriculture* 2010; **74**(1):129–136.
15. Zhao G, Bryan BA, King D, Luo Z, Wang E, Bende-Michl U, Song X, Yu Q. Large-scale, high-resolution agricultural systems modeling using a hybrid approach combining grid computing and parallel processing. *Environmental Modelling and Software* 2013; **41**:231–238.
16. Pannell D. On the estimation of on-farm benefits of agricultural research. *Agricultural Systems* 1999; **61**:123–134.
17. Jones J, Keating B, Porter C. Approaches to modular model development. *Agricultural Systems* 2001; **70**(2-3): 421–443.
18. Folino G, Spezzano G. An autonomic tool for building self-organizing grid-enabled applications. *Future Generation Computer Systems* 2007; **23**:671–679.
19. Thain D, Tannenbaum T, Livny M. Condor and the grid 2003. 299—335.

DOI: 10.1002/cpe

20. Condor team. DAGman: a directed acyclic graph manager. (Available from: http://research.cs.wisc.edu/condor/dagman/dagman.html) [Accessed on May, 2014].
21. Deelman E, Singh G, Su MH, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, Laity A, Jacob JC, Katz DS. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 2005; **13**(3):219–237.
22. Taylor I, Wang I, Shields M, Majithia S. Distributed computing with Triana on the grid: research articles. *Concurrency and Computation: Practice and Experience* 2005; **17**(9):1197–1214.
23. Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock MR, Wipat A, *et al.* Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* Nov 2004; **20**(17): 3045–3054.
24. Cao J, Jarvis SA, Saini S, Nudd GR. Gridflow: workflow management for grid computing. *Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, CCGRID '03, IEEE Computer Society: Washington, DC, USA, 2003; 198–205.
25. Johnson D, Meacham K, Kornmayer H. A middleware independent grid workflow builder for scientific applications. *2009 5th IEEE International Conference on E-Science Workshops*, Oxford, United Kingdom, 2009; 86–91.
26. Altintas I, Berkley C, Jaeger E, Jones M, Ludascher B, Mock S. Kepler: an extensible system for design and execution of scientific workflows. *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004*, Santorini Island, Greece, 2004; 423–424.
27. Kovács M, Gönczy L. Simulation and formal analysis of workflow models. *Electronic Notes in Theoretical Computer Science* 2008; **211**(0):221–230.
28. Goodheart B, Cox J. *The magic garden explained: the internals of UNIX system V release 4: an open systems design*. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1994.
29. Fahringer T, Prodan R, Duan R, Nerieri F, Podlipnig S, Qin J, Siddiqui M, Truong HL, Villazon A, Wieczorek M. Askalon: a grid application development and computing environment. *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, IEEE Computer Society: Washington, DC, USA, 2005; 122–131.
30. Hijma PL, van Nieuwpoort RV, Jacobs CJ, Bal HE. Generating synchronization statements in divide and conquer programs. *Parallel Computing* 2012; **38**(1-2):75–89.
31. Mateos C, Zunino A, Hirsch M, Fernández M. Enhancing the BYG gridification tool with state-of-the-art grid scheduling mechanisms and explicit tuning support. *Advances in Engineering Software* 2012; **43**(1):27–43.
32. Delaittre T, Kiss T, Goyeneche A, Terstyanszky G, Winter S, Kacsuk P. GEMLCA; running legacy code applications as grid services. *Journal of Grid Computing* 2005; **3**(1-2):75–90.
33. Ho QT, Hung T, Jie W, Chan HM, Sindhu E, Subramaniam G, Zang T, Li X. GRASG – a framework for "gridifying" and running applications on service-oriented grids. *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, IEEE Computer Society: Washington, DC, USA, 2006; 305–312.
34. Kommineni J, Abramson D. Griddles enhancements and building virtual applications for the grid with legacy components. *Proceedings of the 2005 European Conference on Advances in Grid Computing*, EGC'05, Springer-Verlag: Berlin, Heidelberg, 2005; 961–971.
35. Johnson R. J2EE development frameworks. *Computer* 2005; **38**(1):107–110.
36. Englander R. *Developing Java Beans*. O'Reilly Media: Sebastopol, CA, USA, 1997.
37. Atkinson M, Deroure D, Dunlop A, Fox G, Henderson P, Hey T, Paton N, Newhouse S, Parastatidis S, Trefethen A, Watson P, Webber J. Web service grids: an evolutionary approach. *Concurrency and Computation: Practice and Experience* 2004; **17**:377–389.
38. Van Nieuwpoort RV, Wrzesińska G, Jacobs CJH, Bal HE. Satin: a high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems* 2010; **32**(3):9:1–9:39.
39. Foster I. Globus Toolkit Version 4: software for service-oriented systems. *Journal of Computer Science and Technology* 2006; **21**:513–520. DOI: 10.1007/s11390-006-0513-y.
40. Deelman E, Gannon D, Shields M, Taylor I. Workflows and e-Science: an overview of workflow system features and capabilities. *Future Generation Computer Systems* 2009; **25**(5):528–540.
41. Romera A, Morris S, Hodgson J, Stirling W, Woodward S. A model for simulating rule-based management of cow-calf systems. *Computers and Electronics in Agriculture* 2004; **42**(2):67–86. DOI: 10.1016/S0168-1699(03)00118-2.
42. Good J, Bright J. An object-oriented software framework for the farm-scale simulation of nitrate leaching from agricultural land uses – IRAP farmsim 2005.
43. Sherlock R, Bright K. An object framework for farm system simulation. In *Proc. MODSIM'99*, Vol. 3, Oxley L, Scrimgeour F, Jakeman A (eds). Modeling and Simulation Society of Australia and New Zealand: Hamilton, NZ, 1999; 783–788.
44. Hillyer C, Bolte J, van Evert F, Lamaker A. The ModCom modular simulation system. *European Journal of Agronomy* 2003; **18**(3-4):333–343.
45. Scott ML. *Programming Language Pragmatics, Third Edition* (3rd edn). Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2009.
46. Altintas I, Wang J, Crawl D, Li W. Challenges and approaches for distributed workflow-driven analysis of large-scale biological data: vision paper. *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, ACM: New York, NY, USA, 2012; 73–78.
47. Qin J, Fahringer T. Advanced data flow support for scientific grid workflow applications. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, ACM: New York, NY, USA, 2007; 42:1–42:12.

DOI: 10.1002/cpe

48. McPhillips T, Bowers S, Zinn D, Ludascher B. Scientific workflow design for mere mortals. *Future Generation Computer Systems* 2009; **25**(5):541–551.
49. Tsangaris MM, Kakaletris G, Kllapi H, Papanikos G, Pentaris F, Polydoras P, Sitaridi E, Stoumpos V, Ioannidis YE. Dataflow processing and optimization on grid and cloud infrastructures. *IEEE Data Engineering Bulletin* 2009; **32**: 67–74.
50. Hartley TD, Saule E, Catalyurek U. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing* 2012; **38**(6-7):289–309.
51. Drejhammar F, Schulte C, Brand P, Haridi S. Flow Java: declarative concurrency for Java. In *Logic Programming, Lecture Notes in Computer Science*, Vol. 2916, Palamidessi C (ed.). Springer Berlin: Heidelberg, 2003; 346–360.
52. Zinn D, Bowers S, Köhler S, Ludäscher B. Parallelizing XML data-streaming workflows via MapReduce. *Journal of Computer and System Sciences* 2010; **76**(6):447–463.
53. Freeh VW. A comparison of implicit and explicit parallel programming. *Journal of Parallel and Distributed Computing* 1996; **34**(1):50–65.
54. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R. Java in the high performance computing arena: research, practice and experience. *Science of Computer Programming* 2013; **78**(5):425–444. Special section: Principles and Practice of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination.
55. Eclipse JDT core. (Available from: http://www.eclipse.org/jdt/core/index.php) [Accessed on May, 2014].
56. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional: Upper Saddle River, NJ, USA, 1994.
57. Mateos C, Zunino A, Hirsch M. EasyFJP: providing hybrid parallelism as a concern for divide and conquer Java applications. *Computer Science and Information Systems* 2013; **10**:1129–1163.
58. Mateos C, Zunino A, Campo M. On the evaluation of gridification effort and runtime aspects of JGRIM applications. *Future Generation Computer Systems* 2010; **26**(6):797–819.
59. Wrzesinska G, van Nieuwport R, Maassen J, Kielmann T, Bal H. Fault-tolerant scheduling of fine-grained task in grid environments. *International Journal of High Performance Computing Applications* 2006; **20**(1):103–114.
60. Gridgain. (Available from: http://www.gridgain.org)[Accessed on September, 2014].
61. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R. Programming, composing, deploying for the grid. *Grid Computing: Software Environments and Tools*, Springer Verlag: New York, USA, 2006; 205–229.
62. Atzeni P, Bugiotti F, Rossi L. Uniform access to NoSQL systems. *Information Systems* 2014; **43**(0):117–133.
63. Berger H. Modelling the effect of maize silage and oat winter forage crop on cow-calf systems in Argentina. *International Grassland Conference. Sydney 15-19 sept*, Sydney, Australia, 2013; 15–19.
64. Wrzesinska G, Van Nieuwpoort R, Maassen J, Bal H. An simple and efficient fault tolerance mechanism for divide-and-conquer systems. *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004*, Chicago, Illinois, USA, 2004; 735.
65. De Boelelaan A, van Nieuwpoort R, Van Nieuwpoort RV, Maassen J, Maassen J, Wrzesinska G, Wrzesińska G, Kielmann T, Kielmann T, Kielmann T, Bal HE, Bal HE. Adaptive load-balancing for divide-and-conquer grid applications. *Journal of Supercomputing* 2004.
66. JPPF. (Available from: http://www.jppf.org/) [Accessed on September, 2014].
67. Gridgain visor. (Available from: http://www.gridgain.com/visor/) [Accessed on October, 2014].
68. Mateos C, Zunino A, Hirsch M. EasyFJP: providing hybrid parallelism as a concern for divide and conquer Java applications. *Computer Science and Information Systems* 2013; **10**:1129–1163.
69. Agrawal D, Das S, El Abbadi A. Big data and cloud computing: current state and future opportunities. *Proceedings of the 14th International Conference on Extending Database Technology,* ACM, Uppsala, Sweden, 2011; 530–533.
70. Lee D, Kim J-S, Maeng S. Large-scale incremental processing with Map Reduce. *Future Generation Computer Systems* 2014; **36**:66–79.

DOI: 10.1002/cpe