Scientific Programming 19 (2011) 161–178
DOI 10.3233/SPR-2011-0321
IOS Press

Using a new event-based simulation framework for investigating resource provisioning in Clouds

Simon Ostermann, Kassian Plankensteiner and Radu Prodan
Institute of Computer Science, University of Innsbruck, Innsbruck, Austria
E-mails: {simon, kassian.plankensteiner, radu}@dps.uibk.ac.at

Abstract. Today, Cloud computing proposes an attractive alternative to building large-scale distributed computing environments by which resources are no longer hosted by the scientists' computational facilities, but leased from specialised data centres only when and for how long they are needed. This new class of Cloud resources raises new interesting research questions in the fields of resource management, scheduling, fault tolerance, or quality of service, requiring hundreds to thousands of experiments for finding valid solutions. To enable such research, a scalable simulation framework is typically required for early prototyping, extensive testing and validation of results before the real deployment is performed.

The scope of this paper is twofold. In the first part we present GroudSim, a Grid and Cloud simulation toolkit for scientific computing based on a scalable simulation-independent discrete-event engine. GroudSim provides a comprehensive set of features for complex simulation scenarios from simple job executions on leased computing resources to file transfers, calculation of costs and background load on resources. Simulations can be parameterised and are easily extendable by probability distribution packages for failures which normally occur in complex distributed environments. Experimental results demonstrate the improved scalability of GroudSim compared to a related process-based simulation approach.

In the second part, we show the use of the GroudSim simulator to analyse the problem of dynamic provisioning of Cloud resources to scientific workflows that do not benefit from sufficient Grid resources as required by their computational demands. We propose and study four strategies for provisioning and releasing Cloud resources that take into account the general leasing model encountered in today's commercial Cloud environments based on resource bulks, fuzzy descriptions and hourly payment intervals. We study the impact of our techniques to the overall execution time, overall cost and cost per unit of saved time with respect to various instance types offered by the Amazon EC2.

Keywords: Grid computing, Cloud computing, simulation, resource provisioning

1. Introduction

Cloud computing is becoming one of the main market-leading trends in the current IT business, emerging as a paradigm for "anything computing" embraced by more and more companies to label their products or ideas. For the lack of a standard definition explaining the term Cloud computing, the interpretations are split in a broad spectrum from leased resources for storage and computation to complete business solutions for enterprises [30]. The scientific community is highly interested in the *Infrastructure as a Service (IaaS)* interpretation characterised by leasing of raw computation, storage, databases, and other resources from specialised providers under certain Quality of Service and Service Level Agreement conditions (usually a certain resource uptime for a certain price). In contrast to other paradigms such as Software as a Service (SaaS) or Platform as a Service (PaaS), IaaS platforms are especially attractive to scientific computing providing opportunities to embed legacy codes in virtual machines, which can be easily deployed and executed on raw Cloud infrastructure resources.

This new class of Cloud resources raises new research questions in the field of resource management (e.g., when to provision new resources, of what type, and at what cost) or scheduling (e.g., how to optimally map applications on existing Cloud resources). Many of these problems are NP-complete and require hundreds to thousands of simulation experiments for validating new approximate solutions. To obtain results in a reasonable timeframe, a scalable simulation framework is typically required for early testing and validation before the real deployment is performed. In general, there are two main advantages of using simulation alongside real executions: (1) they allow investigation of a larger number of parameter configurations, machine sizes, or scenarios that are too difficult or impossible to achieve in practice; (2) the simulation time for testing validating a solution is significantly lower compared to a real execution. In Clouds, the role of a simulator becomes even more important, since computing cost models are an integrated part of any Cloud environment, in contrast to computational Grids where resources are assumed to be freely shared.

In previews work [25], we studied the possibility of extending Grid infrastructures with IaaS Cloud resources to improve the execution of large-scale workflow applications that do benefit from sufficient Grid resources as required by their computational demands. When extending scientific Grids with Clouds, an important issue is to minimize their leasing cost while maximizing the contribution to reducing the overall workflow execution. From this point of view, IaaS Clouds are different than other infrastructures such as business Grids for several reasons. First, Cloud resources are rented by providers in predefined bulks (also called *instances*) and for a fixed duration (usually one hour), each partial interval consumed being billed as full. Second, Cloud providers advertise virtual resources in fuzzy terms, which makes global scheduling optimizations extremely difficult or impossible to make. For example, Amazon Elastic Compute Cloud (EC2) advertises its resources in *EC2 Compute Units (ECU)* representing the speed of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon-*equivalent* processor, which are difficult to map to today's multicores processor and may offer a significantly different performance than expected. In this context, there is a lack of support in the community for scalable and easy to use simulation frameworks able to support combined Grid and Cloud scientific research. Existing simulators such as GridSim [28] and CloudSim [5] follow a *process-based* approach that runs a separate thread for each entity in the system resulting in poor scalability when the number of entities in the system becomes large.

The paper is organized in two parts to address these issues. In the first part, we describe GroudSim, a new an *event-based* simulator for scientific applications on Grid and Cloud environments that requires one simulation thread only (instead of one thread per entity). We present experimental results that demonstrate the scalability of approach with respect to sequential and parallel job submissions and file transfers, as well as the superiority over the process-based approach for simulating the execution of two real-world workflows. As a case study, we use in the second part of the paper our new simulator to investigate the problem of dynamically provisioning of additional Cloud resources to large-scale scientific workflows running in Grid infrastructures with respect to four important aspects: (1) *Cloud start* representing when is sensible to extend the Grid infrastructure with Cloud resources, (2) *instance size* quantifying the amount of Cloud resources that shall be provisioned; (3) *Grid rescheduling* indicating when to move computation from Cloud to the Grid, if new fast resources become available; and (4) *Cloud stop* meaning when it is sensible to release Cloud resources if no longer necessary considering their hourly payment interval. We analyse the impact of these four aspects with respect to the overall execution time, overall cost, as well as cost per unit of saved time for using Cloud resources.

The paper is organised as follows. Section 2 summaries the related work, followed by an introduction to the discrete-event simulation technology. Section 3 presents the GroudSim simulator in detail, including a scalability and a comparative evaluation against other related simulators. In Section 4, we present a case of using our simulator for studying the four dynamic Cloud resource provisioning strategies. Section 5 concludes the paper.

2. Related work

GridSim [28] is a simulation toolkit for resource modelling and application scheduling for Grid computing. GridSim uses SimJava [19] as the underlying simulation framework, which is a process-based discrete event simulation package that runs a separate thread for each entity in the system resulting in poor performance. Evaluation results show that this toolkit suffers when simulating more than 2000 Grid sites concurrently, followed by an "out of memory" error. CloudSim [5] extends GridSim by modelling and simulating Cloud computing infrastructures and services showing the same scalability problems.

SimGrid [6] is a simulation framework for evaluating cluster, Grid, and peer-to-peer algorithms and heuristics. The approach is comparable to the one used in GroudSim, but uses C instead of Java as the main development language, which makes its integration with existing Java tools and services such as the ASKALON

Grid application development and computing environment [11] more difficult. Currently, we are integrating GroudSim into ASKALON to allow the user to easily switch from simulations to real executions from the same unique graphical user interface. Integration of C-based SimGrid into ASKALON using the Java Native Interface technology would break the "compile once, run everywhere" advantage of Java. Our goal is to achieve a simulation framework that can be run on any architecture directly from the browser using Java Webstart technology, which is not possible using the C language that needs to be compiled for each architecture and operating system separately. This and other integration reasons such as user interfaces for configuring the simulation and direct communication within a Java container to eliminate Web services overheads lead us to the decision of developing a new Java-based simulator. Furthermore, since SimGrid does not address simulation of Cloud infrastructures, we were unable to compare its performance with GroudSim.

There are several Grid systems such as Tycoon [17] and Spawn [31] designed to operate using market mechanisms. However, Cloud providers are the first to make computational and storage resources commercially available on pay-per-use basis at production level. The work in [12] compares Grid with Cloud resources and shows the potential for their combination, which the approach presented in this paper, is adapting to provide a hybrid environment using booth resource types by adopting their similarities.

There are a number of important projects showing a growing interest in Cloud computing in the scientific and open source communities. The Nimbus [15] package provides a scientific Cloud middleware deployed in an informal group of four university Clouds called Science Cloud. Hadoop [2] is a toolkit for distributed computing allowing map-reduce applications to be developed and executed in a complete tool chain that supports Cloud resources and Grids. A commercial open-source implementation of a Cloud middleware compatible with EC2 is provided by Eucalyptus [23].

Provisioning of Cloud resources for SaaS architectures is addressed in [20], which is a different method of enabling scientific applications less general and flexible than IaaS (in particular for the deployment of legacy codes), but has other advantages such as easier use and possibly less middleware overheads. The developed methods try to minimise the resource cost for concurrent SaaS executions, which is a different execution scenario that cannot be applied on scientific workflow executions, as analysed in this paper.

In [10], several extensions to BPEL for using Cloud resources in cases of peak load situations are proposed. The approach does not optimise the use of the Cloud resources, nor is well-suited for parallel scientific applications, as one of the constrains of the proposed extensions is that execution servers may handle multiple requests simultaneously. This assumption conflicts with classical scientific computing scenarios, which rely on local resource management systems for exclusive access to processing resources.

The scientific work presented in [27] is comparable to our approach, but focused on fault tolerant scheduling that does not optimise the schedule for Cloud usage and cost. The work in [9] is limited to using Cloud resources for executing an image mosaic workflow, and considers second billing intervals instead of hourly billing intervals that are common in today's commercial Cloud environments, leading to a unrealistic cost analysis.

3. GroudSim

In this section, we describe the technical details of GroudSim by referencing to the most important Java API classes available at [4]. We start by introducing the main concepts of discrete-event simulation, which is the main technology underneath our approach.

3.1. Discrete-event simulation

A discrete system [22] is one in which the state variables change only at discrete points in time called *events*, whose a chronological sequence describe the behaviour of the system. The following terms are important when working with a discrete-event simulation system: (1) *event* being an instant occurrence that changes the state of a system; (2) *future event list* (FEL) being a list of future events that is ordered by their occurrence in time; (3) *clock* being a variable representing the time at which the simulation currently stands; (4) *entity* being any object or component in the system that requires explicit representation in the model; and (5) *system state* being a collection of variables that contain all the information necessary to describe the system at any time. In our case, the system variables are the Grid and Cloud resources which get jobs assigned and, using their computational power, save the future job finish events in the FEL.

Further, a discrete-event simulation system also needs a so-called *time advance algorithm*, which is used to advance the simulation clock when there are no more external requests. An *event scheduling algorithm* is another very important part of a discrete-event simulator, responsible for the correct processing order of events. An event can influence other events that are stored in the FEL that might need to be removed or altered, leading to a critical dependency in the order of event processing.

The main advantage of the discrete-event simulation method is that it makes use of a single thread and is therefore much more scalable, as we will demonstrate in Section 3.10.1. In contrast, existing simulators such as GridSim [28] and CloudSim [5] follow a *process-based* approach that runs a separate thread for each entity in the system resulting in poor scalability when the number of entities in the system becomes large.

3.2. Entities

`SimEngine` is the main GroudSim class which implements the time advance algorithm, the clock, and the FEL, and keeps track of the so-called *registered entities* used for tracing during a simulation. There are three options when starting a simulation: (1) simulate as long as there are events in FEL; (2) simulate for a specified simulation time; and (3) simulate until an arbitrary point in time and shutdown the `SimEngine` afterwards.

The Grid and Cloud resources classes share most of the common functionality implemented in the `groud` package, and override the specialised behaviour in the `groud.grid` and `groud.cloud` packages. To allow manipulation of the state of entities (e.g., `CloudSite`, `GridSite`), a level of indirection for forwarding events directly to the destination entity is added. `GroudEntity` is an abstract class which provides all method stubs for manipulating the state of entities.

Several Cloud instances of the same `InstanceType` can be acquired using one `ResourceReservation` as possible using the Amazon EC2 API to save acquisition time when requesting multiple instances. Each Cloud instance is an object of type `CloudSite` registered properly with the simulation engine. On release of a `CloudSite`, the simulator checks if there are jobs still running which need to be cancelled before the release of the `ResourceReservation` is confirmed. We designed the functionality of the Cloud instances to match the one of Amazon EC2, which is also used by academic and commercial Cloud middlewares such as Eucalyptus [23] and Nimbus [15].

3.3. Jobs

A `GroudJob` has an identifier, a problem size (in million of instructions (MI)), a source (needed for cancelling it), and can be executed on a Grid or a Cloud site. A `GroudJob` also has a state which is changed during the execution of the specific `JobEventTypes`: `unsubmitted`, `submitted`, `queued`, `activated`, `finished`, `failed` and `cancelled`. Grid and Cloud jobs that specialise a `GroudJob` differ in their execution policy. Grids follow a job queuing policy by putting the jobs into a waiting queue until a CPU becomes available. For using a Cloud, resources, also called *instances*, need first to be acquired, after which a resource policy sharing upon job arrival is applied (no queuing mechanism employed). For each state in the job state transition diagram, there exists a corresponding event type in the `groud.event.job` package implementing a callback method on the source of the event, on its destination, or on both. The only classes that the end-user directly needs to use in his simulation are `JobSubmitEventType` for submitting `JobCancelEventType` for cancelling jobs.

Figure 1 shows the interaction of three possible entities: a user, a `SimEngine`, and one `GridSite`. The first step has to be initiated by the user, while the rest are done automatically by GroudSim:

1. The user adds a `JobSubmitEventType` to the `SimEngine`;
2. The `submit` event occurs and the `submitJob` method of the target Grid site is called. The job is in state `submitted`;
3. The Grid site creates a `JobQueuedEventType` and adds it to the `SimEngine`;
4. The `queued` event occurs and the `handleJobQueued` method of both the user and the Grid site is called. The job is in state `queued`;
5. The user needs to implement the `handleJobQueued` method. The Grid site adds `Job-`

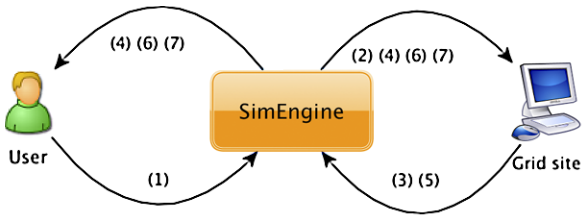


Fig. 1. Job submission workflow. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

ActivatedEventType and JobFinished-EventType events to the SimEngine, as it already knows how much time the job will need to finish;

6. The activated event occurs and the handleJobActivated method of both the user and the Grid site is called. The job is in state activated. The Grid site resets the consumed MI of the job, indicating that it is starting to run;
7. The finished event occurs and the handle-JobFinished method of both the user and the Grid site is called. The job is in state finished. The Grid site releases the CPU occupied by the job and calculates its costs. The user can now analyse the costs, the runtime, or submit new jobs.

3.4. File transfer

Each Groud (i.e., GridSite and CloudSite) has a NetworkLink representing its connection to any other Groud. A FileTransfer knows the size of the file and the bandwidth of the simulated NetworkLink. As two different Grouds may have different bandwidths, the actual bandwidth is the minimum of the available bandwidths of both. The state model is similar to the one of the Groud-Job, except that the state queued is missing. Several event types are handled within the file transfer simulation similar as in the case of Groud-Jobs. The classes the end-user needs to use are FiletransferSubmitEventType and File-transferCancelEventType, and two additional important event types: NetworkLinkFailureET and NetworkLinkRecoveryET.

3.5. Cost

We support two cost models in the simulation environment. For Grid resources, the computation time is typically free but might be limited to using different local queues or charged in service units, similar to the Tera Grid approach. GroudSim collects the used CPU time and can disable Grid resources once their use limit is reached. Cloud instances have to be paid for their usage, typically on an hourly basis as charged by most of today's commercial Cloud providers. GroudSim allows keeping track of the costs resulting from a simulation and supports custom billing intervals to study their influence on the overall cost. The cost introduced by file transfers is calculated per gigabyte of data to allow rich simulation scenarios and detailed analysis on Cloud or mixed resource setups. The end-user can retrieve these costs during runtime to allow steering of scheduling polices or at the end of the simulation for later analysis. The cost resulting from additional storage like Amazon EBS or Azure Blob is currently not calculated by GroudSim as these costs rely on a monthly usage of the storage space. An extension of the simulator to support this additional cost in planed.

3.6. Tracing

Tracing is an essential tool to support the offline evaluation of simulation results. GroudSim provides two different configurable tracing types: (1) *entity state tracing* for analysing the system state of all entities in the current simulation including active entities like GridSites and CloudSites, and passive entities such as users; and (2) *event-based tracing* is based on the simulated events and hence more powerful than the static entity-based tracing. Nevertheless, there are simulation results which are more intuitive to gather with entity-based tracing such as the utilisation of the current Groud entities. We designed a tracing architecture similar to the one used by java.util.logging which includes three important additional classes: (1) Tracer defines the link between the simulation engine and the visualisation of the tracing stream; (2) Handlers are responsible for the visualisation of the current tracing stream, including the writing of information to a console or a tracing file as, well as the creation of predefined charts; and (3) Filters are used to remove unnecessary information from the tracing stream for a specific handler.

3.7. Probability distributions

As GroudSim is based on a time sharing system with a lot of different initial timespans and stochastic decisions, distributions are used at multiple points in a simulation. This affects the runtime and the failure behaviour of Grid sites and Cloud instances, as well as the distribution of the initial jobs simulated. The groud.dist package introduces an adapter pattern to use different stochastic distributions from different packages while providing a homogeneous interface. A wide range of different distributions including the widely-used exponential and logarithmic, as well as simpler distributions such as normal or uniform are included. Our implementation uses the standard ssj.jar stochastic package [18,29], which gives the possibility to run deterministic and nondeterministic simulations by using precise or random initial seeding values.

3.8. Failures

As real Grids and Clouds are distributed systems prone to failures, the simulator provides the possibility to let some of the registered resources fail for certain time intervals. Furthermore, the problem size and the occurrence probability can be configured for each failure, thus tuning the simulation for different degraded levels of service. The simulator provides two different types of failures implemented in the `groud.failure` package: job and file transfer-related. Each `GroudEntity` defines its own failure behaviour. The standard behaviour is configurable via the `GroudSimEntityProp` and follows a stochastic distribution for each failure property. As already mentioned, these properties consist of the size of the failure, the duration of the failure and the mean time to next failure for both jobs and file transfers. For activating the failure behaviour for all registered entities, one has to introduce and register the `GroudFailureGenerator` in the simulation engine. From an abstract point of view, this failure generator is another passive simulation entity registered. The failure generator iterates over all registered entities before the simulation starts and adds one reactivation event to the FEL for each entity. Once the simulation reaches such a reactivation event, the failure generator gets activated and injects a failure with the defined size at the target entity. At the same time, events for recovering the affected entity and for reactivating the failure generator are added to the FEL. Using this "circle" of simulated events, each failure behaviour can be simulated for Grid, Cloud, as well as for network resources.

Currently, we are working towards integration of the Failure Trace Archive [16] within our failure generator by providing an interface to the FTA a standard format for failure traces and injecting failures into the system according to the availability traces of the different distributed systems.

3.9. Background load

GroudSim offers functionality to introduce background load into the current simulation by building an interface to the file format of the Grid Workload Archive [13]. The `BackgroundLoader` class located in the `groud.bg` package contains the main functionality of background loading able to introduce new `GroudEvents` into the `SimEngine` and to handle the generated callbacks for each `GroudJob` executed. For each GWA entry that is properly parsed, the `BackgroundLoader` introduces a new `GroudJob` into the current simulation.

3.10. Evaluation

The purpose of our evaluation was twofold:

(1) to compare the performance of GroudSim [28] with GridSim for simulating real workflows;
(2) to evaluate the scalability of the simulator for a large number of parallel and sequential job submissions and file transfers.

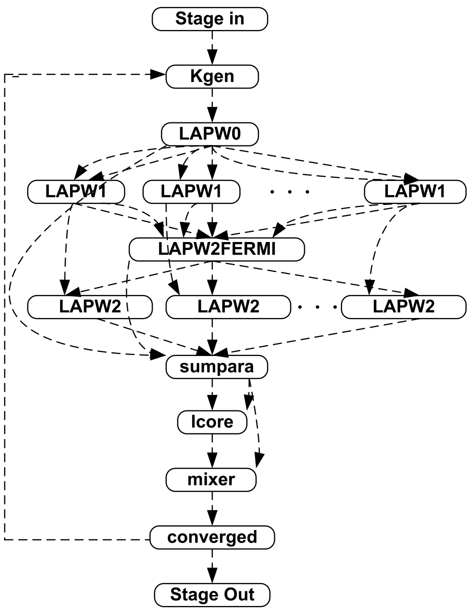
The type of Cloud resources is not relevant to these experiments, as we are only interested in how fast a certain number of jobs on a certain amount of resources can be simulated, independent of their real execution time. We run the GroudSim evaluation experiments on an Intel Core Duo E6750 (2.67 GHz) with 2048 megabytes DDR2-RAM using the Java™ SE Runtime Environment (build `1.6.0_16-b01`). Each experimental result presented represents the average of ten separate runs.

3.10.1. GridSim comparison

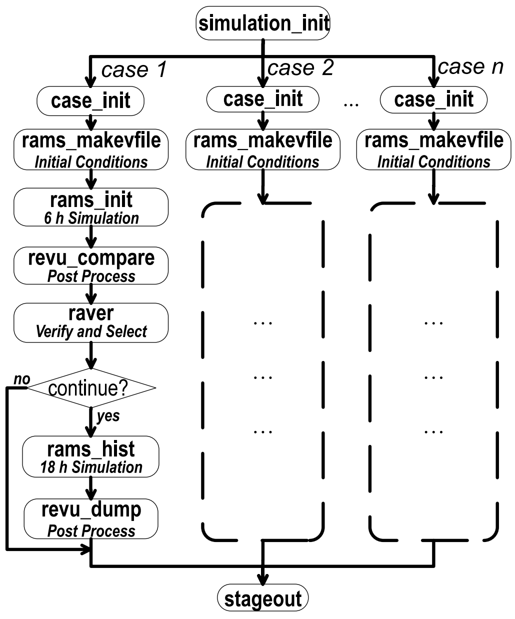
We start our evaluation by comparing GroudSim with the GridSim [28] simulator. We implemented a simple workflow execution environment capable of working with both GridSim and GroudSim as backend and used two real-world workflow applications in our evaluation: WIEN2k and MeteoAG. The size of the simulated workflows can be changed using a parameter x called *parallelization size*, which corresponds to the problem size of the input data and the workflows are executed on a Grid configuration comparable with the Austrian Grid. We generated the performance models for these applications from real trace data logged in the Austrian Grid environment over the course of the last few years.

WIEN2k [3] is a material science workflow for performing electronic structure calculations of solids using density functional theory based on the full-potential (linearised) augmented plane-wave ((L)APW) and local orbital (lo) method (see Fig. 2(a)). The WIEN2k workflow contains two parallel sections of size x, with sequential synchronisation activities in between. The total number of activities in a WIEN2k workflow is: $N_{wien2k} = 2 \cdot x + 3$.

MeteoAG [7] is a workflow designed for meteorological simulations based on the RAMS numerical atmospheric model (see Fig. 2(b)). The simulations produce spatial and temporal fields of heavy precipitation cases over the western part of Austria to resolve most alpine watersheds and thunderstorms. The workflow structure, in which a large set of simulation cases

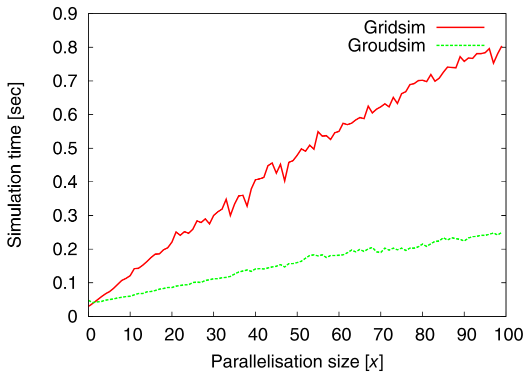


(a)

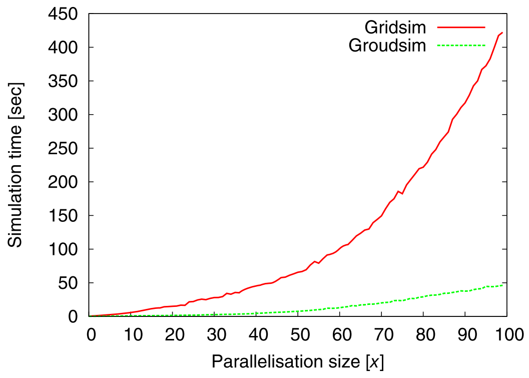


(b)

Fig. 2. Real-world scientific workflow applications. (a) WIEN2k where x is the amount of LAPW1 and LAPW2 tasks. (b) The MeteoAG workflow for $x = n$.



(a)



(b)

Fig. 3. GroudSim and GridSim comparison. (a) WIEN2k; (b) MeteoAG. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

x (parallelization size) is modelled as a parallel loop, where for each simulation, another nested parallel loop is executed with different parameter values. The total number of activities in a MeteoAG workflow is: $N_{meteoag} = 69 \cdot x + 2$.

Figure 3 shows that for growing parallelization sizes, the GridSim simulation time increases significantly faster than the GroudSim for both workflows. The reason for this advantage is in the event-based nature of GroudSim, in which the number of simulated resources (Grid sites, Cloud instances) and tasks has very little impact on the simulation time, as demonstrated in the following next two sections.

3.10.2. Job submission

Figure 4(a) shows the results of the parallel submission of multiple jobs to Grid sites with 32 CPUs each and a computing power of 1000 MI per second (MIPS) for each CPU. We ran the tests on 8 to 32,768 Grid sites and submitted between 16,384 and 1,048,576 jobs. Submitting four times as many jobs to a given number of Grid sites requires four times as long simulation time to complete, slightly longer due to the Java Vir-

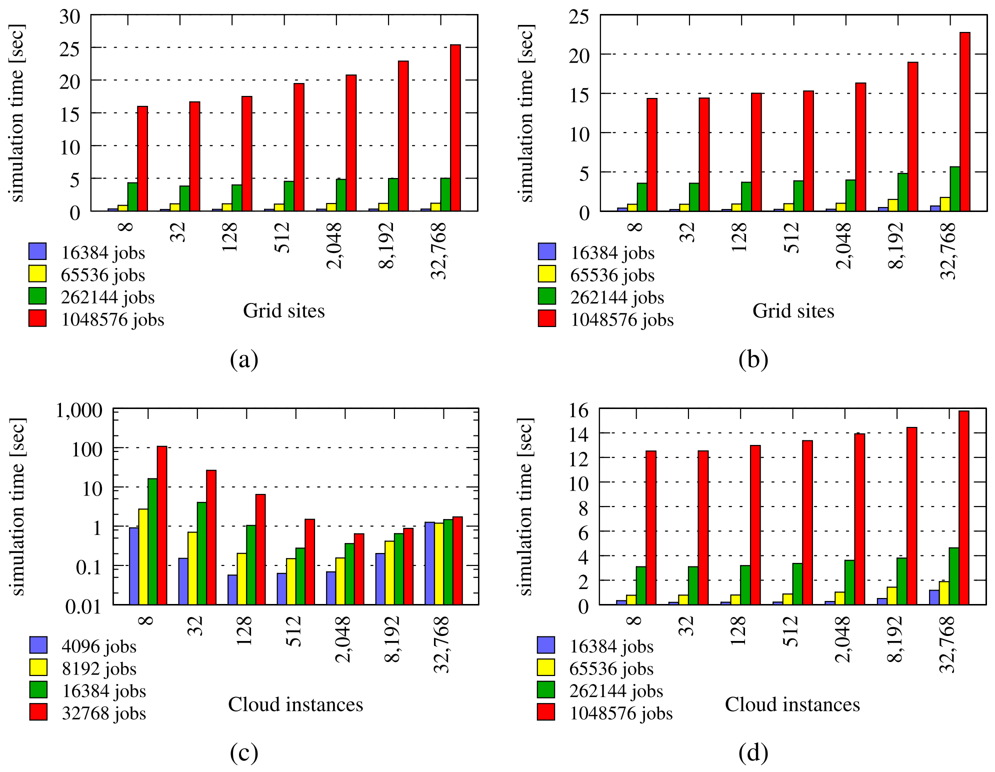


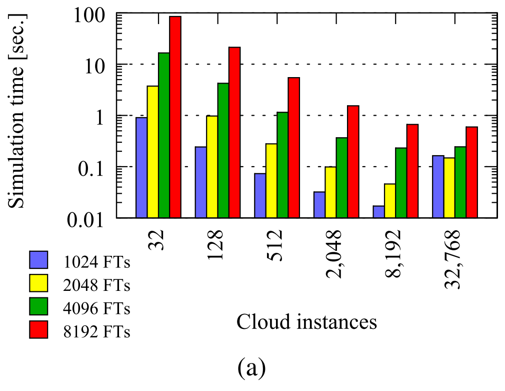
Fig. 4. Job submission experimental results. (a) Parallel Grid job submission. (b) Sequential Grid job submission. (c) Parallel Cloud job submission. (d) Sequential Cloud job submission. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

tual Machine (JVM) garbage collector. The comparison between the clusters shows that the simulation also scales with the number of registered Grid sites, the execution times being almost independent of the number of entities except for cases when the available memory is low. Different amounts of computing power per CPU did not affect the runtime at all, therefore the results of these experiments are not presented. The different number of CPUs per Grid site means the creation of additional objects, however, the overhead caused by this parameter is negligible.

Figure 4(b) illustrates the results for the sequential submission of jobs using the same setup as in the previous experiment. This scenario implies that each Grid site executes jobs in sequence, but the jobs are submitted only when the resources are available. The results are similar to the previous experiment results, the only difference being in simulation time which is slightly shorter due to the smaller amount of events in the FEL and objects in the Java Virtual Machine, resulting in a better overall performance.

Figure 4(c) shows the results for the Cloud parallel job submission for which the performance can be low when the number of acquired instances is small. The reason lies in the nature of event-based simulation and the resource sharing policy of Cloud instances. Whenever a new job is submitted to a CPU of a Cloud instance that has a number of jobs already running, the finish time of all of these jobs needs to be recalculated since they are potentially influenced by the newly submitted job. In terms of implementation, this implies removing all future events from the FEL corresponding to each influenced job (i.e., the finish events), and recalculating and recreating the new finish events. Therefore, the worst case scenario has an exponential runtime which is an unrealistic use-case anyway, as no user would run multiple programs on a single core simultaneously.

Figure 4(d) presents the scenario where the jobs were submitted sequentially, showing that the simulation scales linearly with the number of jobs. Moreover, the number of acquired Cloud instances does not have a significant impact on the simulation time, which gets slightly worse the more Cloud instances are acquired due to the huge amount of objects that need to be managed by the JVM.



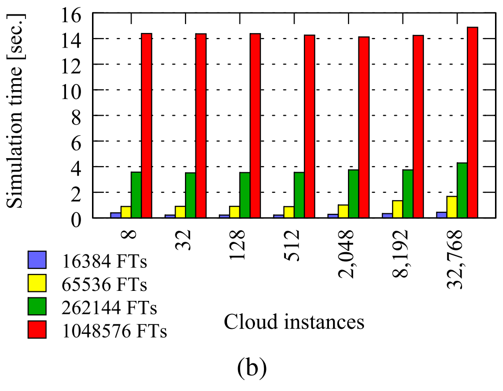


Fig. 5. File transfer experimental results. (a) Parallel Grid file transfers. (b) Sequential Grid file transfers. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

3.10.3. File transfers

Similar to the parallel job submission on Cloud instances, the parallel file transfer results depicted in Fig. 5(a) show an exponential simulation time for the worst case scenario. The reason is similar, file transfers being run on shared network links sharing their bandwidth and influencing each others' events. The finish events of other file transfers on the same network link need therefore to be removed, recalculated, and reinserted into the FEL. However, compared to the corresponding experiments for jobs on Cloud instances (see Fig. 4(c)), the performance is worse because one file transfer involves two network links for being transferred compared to one Cloud resource for computational jobs.

Figure 5(b) illustrates a linear scaling of the number of sequential file transfers and the fact that an increasing number of entities does not significantly influence the results.

4. Optimised Cloud provisioning

As a case study, we illustrate in this section the use of GroudSim to investigate different resource provisioning strategies for scientific workflows that can benefit from additional Cloud resources if there are not sufficient Grid resources to support their computational requirements.

We consider scientific workflows as a set of legacy codes called *activities* interconnected in a directed graph through control flow and data flow dependencies. To support this scenario, we propose several provisioning strategies for Cloud resources to scientific workflows with awareness of the minimum allocation granularity (typically one hour per set of cores). For this purpose, we define a new metric C_T called *cost per unit of saved time (\$/T)* as ratio between the total cost $Cost_{Cloud}$ of leasing the Cloud resources and the time saved by using them:

$$C_T = \frac{Cost_{Cloud}}{T_{Grid} - T_{Grid+Cloud}}.$$

The terms T_{Grid} and $T_{Grid+Cloud}$ can be calculated or approximated in three different ways: real execution, simulation or using prediction models. The goal of the dynamic provisioning is to maximise this metric by increasing the amount of saved time and reducing cost of Cloud resources used.

We assume a just-in-time workflow scheduling approach (as opposed to complete full-ahead scheduling), in which each activity ready for execution is submitted on the fastest available resource (core) expected to deliver the earliest completion time. We opted for a just-in-time approach for several reasons: (1) the lack of accurate performance models on virtual Cloud resources with approximate characteristics may cancel any global optimisation attempts; (2) a small number of Cloud resources (i.e., maximum of eight cores on the two largest Cloud providers [1,26]) share in general one file system compared to Grid environments hosting large clusters with hundreds of cores and one network file system, which increases the amount of file transfer and introduces additional communication overhead; (3) it gives opportunities at every scheduling step to decide on whether to use additional or less Cloud resources to improve the execution.

To address this third goal, we propose in this paper four dynamic resource provisioning techniques described in the following sections: Cloud start, instance size, Grid rescheduling and Cloud stop.

4.1. Cloud start

In our model, an important task of the scheduler is to dynamically complement the Grid infrastructure with additional Cloud resources during runtime if this presents potential for accelerating the workflow execution. In our case, this happens when the amount of Grid resources are insufficient for executing large workflow parallel regions that need to be serialised (see Fig. 6(a)). In doing so, the scheduler has to take two important decisions such that the cost per unit of saved time is maximised: (1) which instance type to lease, and (2) the number of such instances.

The scheduler decides to use Cloud resources if they are faster than the slowest Grid processors, including the estimated resource acquisition and data transfer times, since an improvement in the execution time is expected (see Algorithm 1, line 2). Slow Cloud instances like `m1.small` offered by Amazon EC2 might introduce a severe load imbalance that slows down the workflow execution instead of improving it, as shown in Fig. 6(b). The scheduler applies the minimum completion time (MCT) algorithm to estimate the execution time of the workflow activities that are ready for execution, for example by using analytical prediction models based on historical information [21]. The expected resource acquisition researched in [14] and the file transfer times are considered as additional overheads in the MCT estimation. A resource request is only considered if the estimated Cloud-enabled execution time is lower than in the case of using Grid resources only. Otherwise, the scheduler waits for new resources to become available which happens once other jobs finish or resource availability changes (line 4).

4.2. Instance size

After deciding to acquire Cloud resources, the scheduler must start a number of instances sufficient to accommodate all activities ready for execution. However, Cloud providers often lease resources in bulks (e.g., certain amount of cores for a certain time interval) and therefore, an important question is whether to round up or down the number of leased instances in case the bulk size is not a factor of the total number of required cores (see Algorithm 2). We designed two strategies to address this issue: (1) *generous* uses the more expensive rounding up solution which gives a higher speedup potential since more resources than necessary are allocated; and (2) *economical* uses a smaller amount of resources which reduces costs but introduces overheads such as serial execution of parallel tasks on certain resources (see Fig. 6(a), often followed by load imbalance as shown in Fig. 6(b)), since less resources than necessary are allocated. We will analyse the tradeoff between these two strategies in Section 4.5.

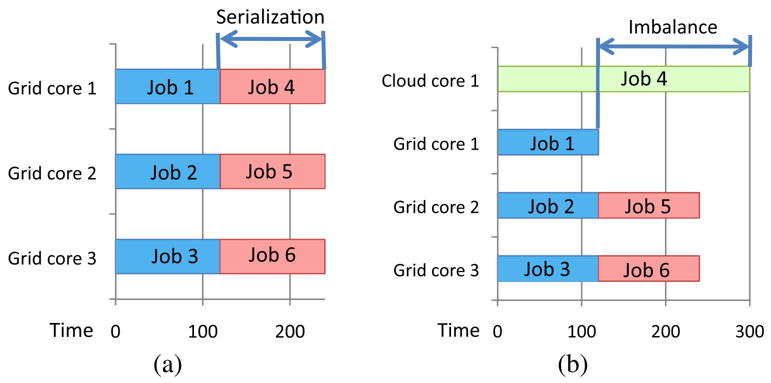


Fig. 6. Serialisation and load imbalance overheads. (a) Serialisation of independent activities on fast resources. (b) Slowdown due to load imbalanced on slow resources. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

Algorithm 1 Cloud start

```
1: no more free Grid resources are available
2: if MCT(ready activities, Grid) > MCT(ready activities, Grid, Cloud)
3:   then startClouds
4: else wait for jobs to finish or resources to become available
```

4.3. Grid rescheduling

Additionally, there may appear special cases during workflow execution when additional shared Grid resources become available, for example because of external jobs completing their execution or simply due to their availability schedule. In these cases, there may be faster or economically more convenient if jobs running on Clouds are stopped and resubmitted to the newly available Grid resources.

Algorithm 3 illustrates the technical implementation of this technique, where the boolean variable `preferGrid` is set by the user to indicate his intentions to save costs (which might increase the execution time). In order to be effective, this method requires a prediction [21] and logging mechanism to estimate the percentage of work completed by the activity running on the Cloud, and whether a reschedule on the Grid is profitable. If `preferGrid` is set to false, the following test is performed for each job running on Cloud resources: if the speed of the new Grid resource (in ECU) is larger than the ratio between the Cloud speed and the percentage of the activity that still needs to be executed (line 3), the activity is resubmitted to the new Grid site where it is expected to complete faster than the instance already running on the Cloud (line 4). Once the resubmitted job is running, the execution on the Cloud is terminated (line 5). Whether the unused Cloud resource is released or not depends on the next billing period, as described in Section 4.4.

4.4. Cloud stop

Once Cloud resources are in use, the next step in the dynamic provisioning is to stop them as early as possible to save money, and to reschedule activities to free Grid resources, if costs can be reduced and the execution time kept unchanged.

Even though the Cloud instances should be stopped as soon as possible if they are not needed by the next activities to be executed in the workflow, there is no benefit of releasing a resource just seconds after its new (hourly) accounting period has started. Instead of being stopped instantly after they become unused, these resources are kept running until the current payment interval gets close to an end, since there may be activities later in the workflow (not yet ready to be started because of dependencies with not yet completed activities) that may use them. Such reuse of Cloud resources may avoid additional deployment and startup overheads as illustrated in Fig. 7 (i.e., “Requested”, “Started” and “Running” periods). The “Accessible” period is extended until the resource uptime

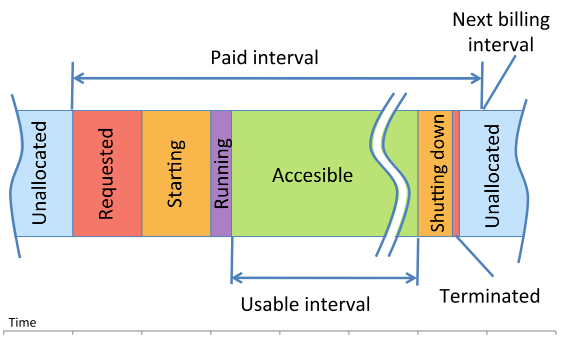


Fig. 7. Cloud resource instantiation and release states (startup and shutdown shown on a nonproportional scale). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

Algorithm 2 Instance size

```
1: scheduler requests n Cloud cores
2: if Cloud provides multiple cores per instance then
3:   n = n / cores per instance
4:   if getLess then n = Math.floor(n)
5:   else n = Math.ceil(n)
6: start n Cloud instances
```

Algorithm 3 Grid rescheduling

```
1: new Grid cores become available
2: for all jobs running on Cloud:
3:   if preferGrid or (GridECUs > (CloudECUs / percentage of job execution left))
4:     then resubmit Cloud job to Grid
5:     cancel job on Cloud
```

Algorithm 4 Cloud stop

```
1: scheduler starts Cloud resources
2: scheduler starts timer with (billingPeriod * 0.95)
3: on timer:
4:   if Cloud resources is not idle then
5:     next timer trigger with (billingPeriod)
6:   else release Cloud resources
```

approaches the next payment interval. Although this additional idle time might reduce the overall Cloud utilisation, it does not increase the cost per unit of saved time. In the ideal case when resources are reused, this ratio may even increase if the saved time leads to less hours of paid Cloud resources. Still, when resources are idle and approach the next payment interval, they need to be stopped in time, as also the shutdown and resource release times are billed.

Algorithm 4 shows the pseudo-code that implements this technique. For each Cloud resource request, which may start one or many instances, a timer is started with the period 95% of the payment interval (line 2) to account for the instance shutdown and release times (see Fig. 7). Once the timer expires, the algorithm checks whether there are activities running on the Cloud associated with this timer and, if so, extends the new timer with a new full billing period. Otherwise it releases the Cloud resources and stops the timer.

4.5. *Evaluation*

For evaluation, we implemented the provisioning methods on top of the GroudSim simulator. As far as the hardware infrastructure is concerned, we chose to simulate three sites available in the Austrian Grid and three instance types offered by Amazon EC2 (see Table 1). We used ECU to characterise the speed of each resource, which we computed by executing the HPL benchmark and normalising the result against the HPL performance of a 1.2 GHz 2007 Opteron processor (see [24]).

Alongside WIEN2k and MeteoAG, we decided to use for the validation a third application called Invmod (see Section 4.5.2) due to the nature of the evaluation results obtained that required an additional application with an intermediate number of activities between WIEN2k and MeteoAG. Each workflow is characterised by a parameter x defining the "parallelization size", which is proportional to the total number N of activities in the workflow. We used this parameter to simulate workflows of different sizes (from small to very large) using historical trace information originating from real executions in the Austrian Grid environment. We first simulated the workflows in a pure Grid environment and then added on-demand Cloud resources to investigate the effectiveness of our dynamic Cloud provisioning strategy with respect to execution time and cost per unit of saved time.

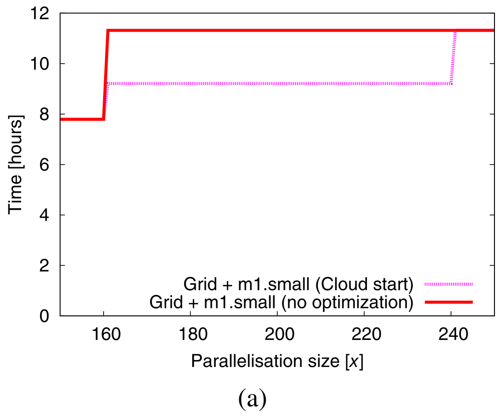
Table 1
The Austrian Grid and Cloud resource testbed

Grid site/ Instance	Location	Cores	Cores per instance	Speed (ECU)	Cost ($/hour)
karwendel	Innsbruck	80	–	2.5	–
altix1.uibk	Innsbruck	16	–	1.5	–
altix1.jku	Linz	64	–	2.0	–
m1.small	Amazon EC2	80	1	1.0	0.095
m1.large	Amazon EC2	80	2	2.0	0.38
c1.xlarge	Amazon EC2	80	8	2.5	0.76

4.5.1. *WIEN2k*

We first analyse the benefits of the dynamic provisioning of Cloud resources for the WIEN2k application introduced in Section 3.10.1, and then the impact of provisioning different types of Cloud instances to the time and cost of execution for different workflow parallelization sizes.

First, using additional Cloud resources can lead to a slowdown if their speed is not taken into consideration, as explained in Section 4.1 (see Algorithm 1). Figure 8(a) shows two scheduling options for parallelization size 160 for which all Grid resources are exhausted: serialise parallel activities on the fastest Grid site `karwendel`, or request slow `m1.small` Cloud resources. We observed in this experiment that requesting slow `m1.small` Cloud resources with no regard to their speed introduces a slowdown through load imbalance, which is much larger than the serialisation overhead present when utilising the Grid environment only. This slowdown is eliminated starting with the parallelization size 240 when the Cloud start-aware scheduler is constrained to utilise Cloud resources because of the large number of workflow activities, which



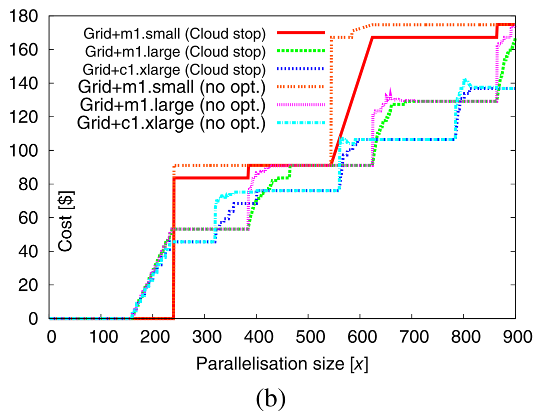


Fig. 8. WIEN2k simulation results for different dynamic Cloud provisioning methods. (a) Cloud start provisioning. (b) Cloud stop provisioning. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

Table 2

Economic versus generous instance type provisioning

Parallel. size [x]	Generous C_T ($/min)	Economic C_T ($/min)
161–167	0.0545	∞
168	0.0545	0.0545
169–175	0.109	∞
176	0.109	0.109
177–183	0.162	∞
184	0.162	0.162
185–191	0.216	∞
192	0.216	0.216
193–199	0.27	∞
200	0.27	0.27
...	...	...
233–239	0.54	∞
240	0.54	0.54

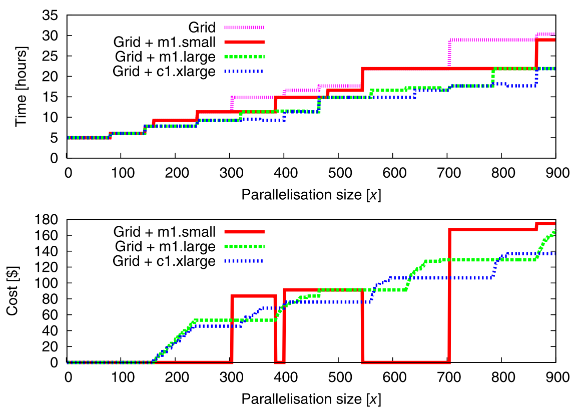
introduces Grid serialisation overheads comparable to the Cloud load imbalance.

Second, Table 2 compares the economical and generous instance size provisioning methods, introduced in Section 4.2 (see Algorithm 2). The economical method requests new `m1.large` and `c1.xlarge` resources only when enough activities are available to entirely fill them, leading to the same execution times as if no Cloud resources are used (because of load imbalance) and, therefore, to an infinite cost per unit of saved time. The generous provisioning shows a more stable step function behaviour matched by the economic provisioning only in the case when the parallelization size is a multiple of the instance size (i.e., eight in case of `c1.xlarge` instances).

Finally, Fig. 8(b) shows the influence of the Cloud stop optimisation described in Section 4.4 for the three Cloud instance types (see Algorithm 4). In most cases, this technique saves costs compared to an immediate resource release scenario. The best improvement is achieved for a parallelization size of 545 and the `m1.small` resources for which the cost is reduced by 45% from $167.2 to $92.15 with no change in execution time.

In the remainder of this section, we analyse the time and cost benefits of dynamic Cloud provisioning to 900 different WIEN2k parallelization sizes (see Fig. 9(a)). In the top diagram, we can observe that the time follows a step function, because the number of cores used in each run is equal to the parallelization size x of the two parallel sections of the workflow. The first steps occur at parallelization sizes 80 and 144, when the three Grid sites are used in the execution environment. The second step is clearly larger than the first because the `altix1.uibk` site is significantly slower than the first two Grid sites. At parallelization size 160, the Grid environment receives additional `m1.large` or `c1.xlarge` Cloud resources, while the `m1.small` instances are added only starting with the 240 parallelization size to avoid load imbalance. The cost curve follows a mixed stepped and linear cost, as shown in the bottom chart. The `c1.xlarge` instances show the best overall performance for this workflow type and also the lowest overall costs, even when compared to the cheaper `m1.small` resources that have the worst performance.

Figure 9(b) shows the costs for saving one minute of execution time when using Clouds. For most parallelization sizes, the `m1.large` instance shows similar performance to `c1.xlarge`, but overall the faster and more expensive `c1.xlarge` resource is the best choice.



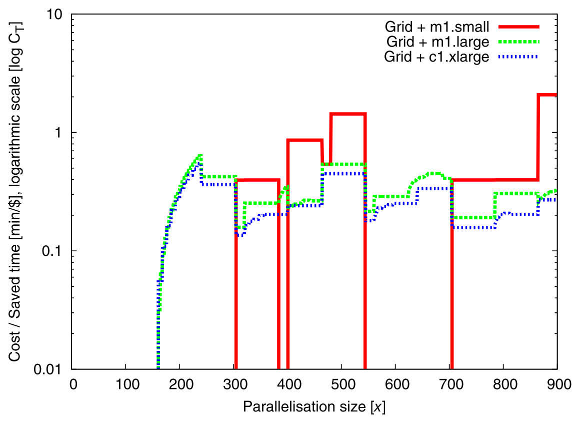


Fig. 9. Simulation results for different WIEN2k parallelization sizes and instance types. (a) Execution times and costs. (b) Cost per unit of saved execution time. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

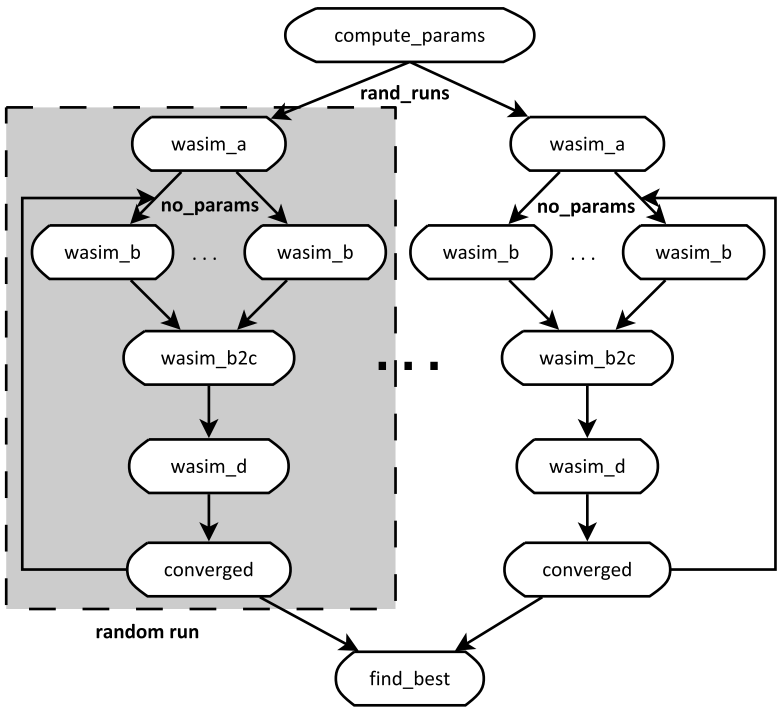


Fig. 10. The Invmod workflow with $x = \mathit{rand\_runs}$ parallelization size.

4.5.2. *Invmod*

Invmod [8] is a hydrological application that uses the Levenberg–Marquardt algorithm to minimise the least squares of the differences between the measured and the simulated runoff for a determined time period. The Invmod workflow displayed in Fig. 10 consists of two levels of parallelism: (1) the outermost parallel loop consists of a number of random runs x (parallelization size) that perform a local search optimisation (in a sequential loop) starting from a random initial solution; (2) alternative local changes are examined for each calibrated parameter in parallel in the inner nested parallel loop. The total number of jobs in an Invmod workflow is: $N_{invmod} = 12 \cdot x + 1$.

Invmod is a workflow with a higher level of parallelism than WIEN2k, whose execution time for 300 different parallelization sizes (between 13 and 3601 activities) is shown in Fig. 11(a) (top). For low par-

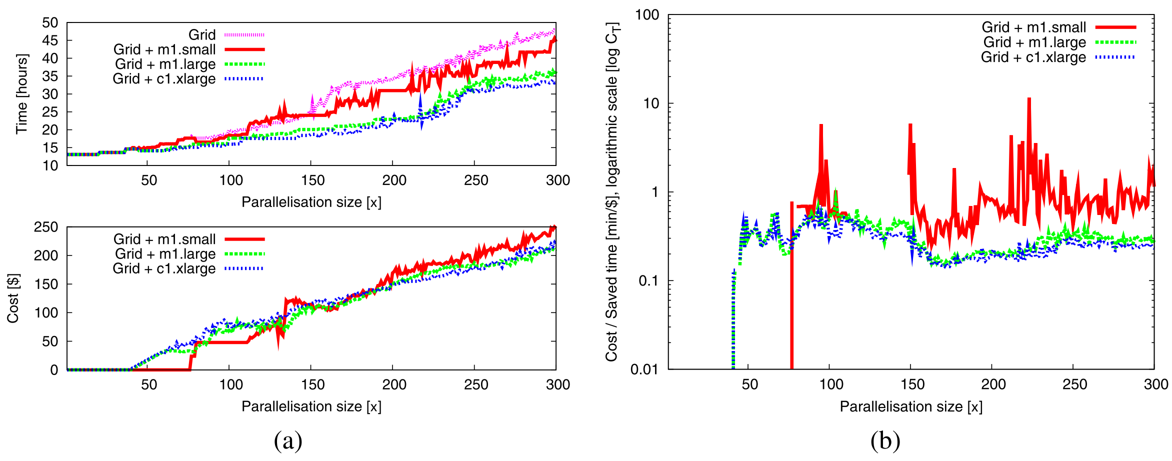


Fig. 11. Simulation results for different Invmod parallelization sizes and instance types. (a) Execution times and costs. (b) Cost per unit of saved execution time. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

allelization sizes between 1 and 50, the execution times and costs show the same characteristics as the WIEN2k. Once the workflow is large enough to occupy the full Grid, the scheduler decides to use Cloud resources starting with $x = 99$ for `m1.small` and $x = 41$ for the faster instance types. The bottom chart shows that the costs for using `m1.small` resources are the lowest for smaller workflows, due to the fact that only a small number of such instances are used because of their low performance compared to the fast Grid resources. Starting with $x = 160$, the Grid resources are overwhelmed by the large number of activities and more `m1.small` instances need to be acquired, which increases the costs over those required by the more cost effective `m1.large` and `c1.xlarge` instances.

Figure 11(b) shows that the costs per unit of saved time for `c1.xlarge` instances are slightly better than for `m1.large`, and significantly better than for `m1.small`, where the gaps in the chart are representing negative values or no saved time, which represent cases in which Cloud resources did not improve the execution. This metric has a high variance for small workflows with parallelization size below 50 in case of `m1.large` and `c1.xlarge` instances, and 250 in case of `m1.small` instances. Large workflows are best supported by fast and more expensive Cloud resources, which provide a stable cost per unit of saved time of \$0.27 for `m1.large` and \$0.24 for `c1.xlarge` instances, compared to \$0.76 for `m1.small` instances.

4.5.3. *MeteoAG*

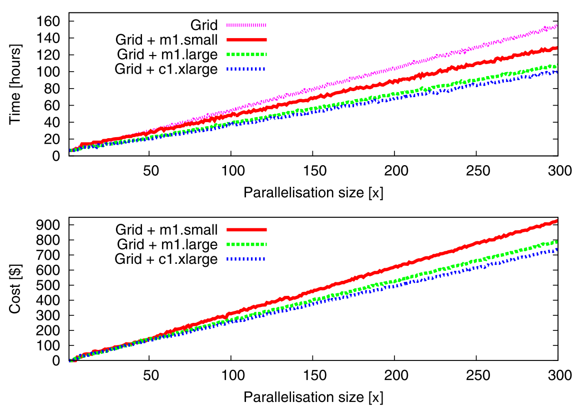
MeteoAG introduced in Section 3.10.1 is a massively parallel workflow with a large number of activities (i.e., between 71 and 20,702) for which the execution time increases linearly with the parallelization size (see Fig. 12(a)). Also for this workflow, the `c1.xlarge` instance delivers for all 300 parallelization sizes the fastest, as well as the cheapest executions, followed by `m1.large` and `m1.small` instances.

The `m1.small` instance has again the highest cost per unit of saved time, while for parallelization sizes below 50 there is no improvement for adding Cloud resources (see Fig. 12(b)[1]). Starting with $x = 100$, we exhibit a nearly constant cost per unit of saved time of \$0.62 for `m1.small`, \$0.28 for `m1.large`, and \$0.22 for `c1.xlarge` instances.

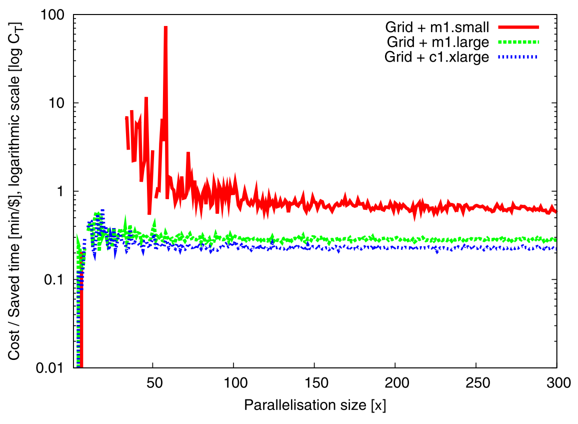
4.5.4. *Summary*

Table 3 summarises the optimised Cloud provisioning experiments conducted with GroudSim for each workflow application. We conducted a total of 12 thousands workflow executions totalising 157 thousands processing hours with an estimated cost of over 642 thousands USD on Amazon `m1.small`, `m1.large`, and `c1.xlarge` instances. To further illustrate the importance of the GroudSim simulator in our study, we use the experimental results from Section 3.10.1 to estimate the time required to conduct the same amount of experiments using GridSim. The simulations needed for our evaluation would have taken over 500 h using GridSim, while using our GroudSim gave us the opportunity to complete the study in less than 60 h on

[1]For small workflow sizes there are a few cases when using Clouds produces no improvement in execution time, which results in a negative or infinite C_T value that cannot be represented at a log scale in Fig. 12(b).



(a)



(b)

Fig. 12. Simulation results for different MeteoAG parallelization sizes and instance types. (a) Execution times and costs. (b) Cost per unit of saved execution time. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0321.)

Table 3

GroudSim versus GridSim simulation summary

Workflow application	Number of executions	GroudSim time (h)	Estimated GridSim time (h)	Estimated execution time (h)	Estimated execution cost ($)
WIEN2k	9600	2.31	8.06	52,074.1	181,401.0
Invmod	1200	0.67	4.11	28,704.7	89,597.2
MeteoAG	1200	56.90	495.62	77,121.8	371,240.0
Total	12,000	59.89	507.80	157,901	642,238

an AMD Opteron 885 with a 2.6 GHz processor, representing a speedup of 8.4. Furthermore, this ratio increases the larger the workflows get, as our scalability experiments from Section 3.10.1 showed.

5. Conclusions

We presented GroudSim, a Java-based simulation toolkit for scientific applications running on combined Grid and Cloud infrastructures. GroudSim uses a discrete-event simulation toolkit that offers improved performance against other process-based approaches used in related work. The developed simulation framework supports modelling of Grid and Cloud computational and network resources, job submissions, file transfers, as well as integration of failure, background load, and cost models. A sophisticated textual and visual tracing mechanism and a library-independent distribution factory give extension possibilities to the simulator: a new tracing mechanisms can be easily added by implementing new handlers or filters in the event system, and additional distribution functions can be included by adding a new library and writing an appropriate adapter. We provided experimental results that demonstrate the scalability of the job submission and file transfer mechanisms, as well as the superiority of our solution over a related process-based approach for simulating the execution two real-world scientific workflow applications.

Using this simulator, we addressed the problem of dynamic provisioning of Cloud resources to scientific workflows that do not have sufficient Grid resources to support their execution. In this context, we analysed the impact of four important Cloud provisioning aspects to the execution time and cost of three real workflow applications in the Austrian Grid combined with Amazon EC2 instances: Cloud start, instance size, Grid rescheduling and Cloud stop. The choice of the correct Cloud instance type is critical: while cheaper resources might look attractive, their slow characteristics degrade the performance in such a way that the overall costs are higher. We further observed that Amazon's `c1.xlarge` instances offer the best price-performance ratio for scientific applications, followed by `m1.large` and `m1.small`. A generous provisioning strategy that allocates more resources than necessary because of the instance bulk renting granularity is more beneficial than an economic strategy, which may degrade performance through serial execution of independent activities that increases costs. Releasing Cloud resources at the end of the billing interval can bring significant cost improvements with no change in execution time, especially in case of slow Cloud resources like the Amazon `m1.small` instance. For relatively small workflows, the execution time and costs follow a step function, as more Cloud resources are provisioned to support larger parallelization sizes. The step function smoothes into a linear function for large workflows with tens of thousands of activities, as the Cloud can be used more uniform. While this may lead to the impression that the Cloud resources might be to expensive to support the execution of large workflows, a more detailed analysis indicated that one minute of saved execution costs between \$0.24 for fast `c1.xlarge` instances and \$0.76 for slow `m1.small` resources. Although this ratio has a high variance for small workflows and slow Cloud resources, it meets the requirements of scientific simulations by showing a very stable and predictable value, the larger the workflows get and the faster the Cloud instances are.

In future work, we plan to use the results from this paper for developing analytical models that characterise the provisioning of Cloud resources and the underlying overheads. We intend to integrate these models into a sophisticated heuristic-based scheduler for full-ahead mapping of workflows in combined Grid and Cloud-based environments. The scheduler will take data locality into account, including the possibility of using Cloud storage resources such as Amazon S3 to compensate the lack of shared file systems in Cloud environments.

The GroudSim framework is integrated as a backend in the ASKALON Grid computing environment, which enables to perform both real and simulated executions of real-world applications using the same integrated development, monitoring, and analysis interface.

References

[1] Amazon, Elastic Compute Cloud (EC2), May 2010, available at: http://aws.amazon.com/ec2/.

[2] Apache, Apache Hadoop project develops open-source software for reliable, scalable, distributed computing, May 2010, available at: http://hadoop.apache.org/.

[3] P. Blaha, K. Schwarz and J. Luitz, *WIEN2k, a Full Potential Linearized Augmented Plane Wave Package for Calculating Crystal Properties*, TU Wien, Vienna, Austria, 2001.

[4] D. Bodner, G. Krahler and S. Joerer, GroudSim Java documentation, 2010, available at: http://www.assembla.com/code/groudsim/subversion/node/blob/trunk/doc/index.html.

[5] R. Buyya, R. Ranjan and R.N. Calheiros, Modeling and simulation of scalable cloud computing environments and the CloudSim toolkit: Challenges and opportunities, in: *7th High Performance Computing and Simulation Conference*, Leipzig, Germany, IEEE, 2009.

[6] H. Casanova, Simgrid: A toolkit for the simulation of application scheduling, in: *First IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, Brisbane, Australia, May 15–18, 2001, IEEE Computer Society, 2001, pp. 430–441.

[7] W.R. Cotton, R.A. Pielke, R.L. Walko, G.E. Liston, C.J. Tremback, H. Jiang, R.L. McAnelly, J.Y. Harrington, M.E. Nicholls, G.G. Carrio and J.P. McFadden, RAMS 2001: Current status and future directions, *Meteorology and Atmospheric Physics* **82** (2003), 5–29.

[8] J. Cullmann, V. Mishra and R. Peters, Flow analysis with WaSiM-ETH – model parameter sensitivity at different scales, *Advances in Geosciences* **9** (2006), 73–77.

[9] E. Deelman, G. Singh, M. Livny, J.B. Berriman and J. Good, The cost of doing science on the cloud: the Montage example, in: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008*, Austin, TX, USA, November 15–21, 2008, IEEE Press, Piscataway, NJ, USA, 2008, p. 50.

[10] T. Dörnemann, E. Juhnke and B. Freisleben, On-demand resource provisioning for BPEL workflows using Amazon's elastic compute cloud, in: *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009*, Shanghai, China, May 18–21, 2009, F. Cappello, C.-L. Wang and R. Buyya, eds, IEEE Computer Society, Washington, DC, USA, 2009, pp. 140–147.

[11] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.L. Truong, A. Villazón and M. Wieczorek, *Workflows for e-Science. Scientific Workflows for Grids*, Springer, 2005, pp. 122–131, Chapter: ASKALON: a Grid application development and computing environment.

[12] I.T. Foster, Y. Zhao, I. Raicu and S. Lu, Cloud computing and grid computing 360-degree compared, in: *Grid Computing Environments Workshop, 2008, GCE'08*, Honolulu, HI, USA, 2008.

[13] A. Iosup, The grid workloads archive, *Future Generation Computer Systems* **24**(7) (2008), 672–686.

[14] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer and D. Epema, Performance analysis of cloud computing services for many-tasks scientific computing, *IEEE Transactions on Parallel and Distributed Systems* (2010), Special Issue on Many-Task Computing, to appear.

[15] K. Keahey, T. Freeman, J. Lauret and D. Olson, Virtual workspaces for scientific applications, in: *Scientific Discovery through Advanced Computing*, Boston, June 2007.

[16] D. Kondo, B. Javadi, A. Iosup and D. Epema, The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems, in: *International Symposium on Cluster Computing and the Grid*, Melbourne, Australia, IEEE Computer Society, 2010.

[17] K. Lai, B.A. Huberman and L.R. Fine, Tycoon: A distributed market-based resource allocation system, Computing Research Repository (CoRR), cs.DC/0404013, 2004.

[18] P. L'Ecuyer, L. Meliani and J. Vaucher, SSJ: a framework for stochastic simulation in Java, in: *WSC'02: Proceedings of the 34th Conference on Winter Simulation*, Winter Simulation Conference, San Diego, CA, USA, IEEE Computer Society, 2002, pp. 234–242.

[19] R. McNab and F.W. Howell, Using java for discrete event simulation, in: *Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW)*, Univ. of Edinburgh, Edinburgh, UK, 1996, pp. 219–228.

[20] R. Mietzner and F. Leymann, Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for SaaS applications, in: *Services I*, IEEE Computer Society, 2008, pp. 3–10.

[21] F. Nadeem, M. Yousaf, R. Prodan and T. Fahringer, Soft benchmarks-based application performance prediction using a minimum training set, in: *International Conference on e-Science and Grid Computing*, IEEE Computer Society, 2006.

[22] B. Nelson, J. Banks, J. Carson and D. Nicol, *Discrete-Event System Simulation*, Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[23] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, Eucalyptus: A technical report on an elastic utility computing architecture linking your programs to useful systems, Technical Report 2008-10, UCSB Computer Science Technical Report, 2008.

[24] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer and D. Epema, A performance analysis of ec2 cloud computing services for scientific computing, in: *Proceedings of Cloudcomp 2009*, October 2009.

[25] S. Ostermann, R. Prodan and T. Fahringer, Extended grids with cloud resource management for scientific computing, in: *Grid09: IEEE/ACM International Conference on Grid Computing*, Banff, Canada, October 2009, pp. 42–59.

[26] Rackspacecloud, Cloud hosting products-using the power of cloud computing by rackspace, May 2010, available at: http://www.rackspacecloud.com/.

[27] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T.M. Huang, K. Thyagaraja and D. Zagorodnov, VGrADS: enabling e-science workflows on grids and clouds with fault tolerance, in: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009*, Portland, OR, USA, November 14–20, 2009, ACM, New York, NY, USA, 2009.

[28] A. Sulistio, U. Cibej, S. Venugopal, B. Robic and R. Buyya, A toolkit for modelling and simulating data grids: an extension to GridSim, *Concurrency and Computation: Practice and Experience* **20**(13) (2008), 1591–1609.

[29] University of Montreal DIRO, Stochastic simulation in Java, available at: http://www.iro.umontreal.ca/~simardr/ssj/indexe.html.

[30] L.M. Vaquero Gonzalez, L. Rodero Merino, J. Caceres and M. Lindner, A break in the clouds: towards a cloud definition, *Computer Communication Review* **39**(1) (2009), 50–55.

[31] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart and W.S. Stornetta, Spawn: A distributed computational economy, *IEEE Transactions on Software Engineering* **18**(2) (1992), 103–117.