J Supercomput
DOI 10.1007/s11227-012-0748-z

Online execution time prediction for computationally intensive applications with periodic progress updates

Maria Chtepen · Filip H.A. Claeys · Bart Dhoedt · Filip De Turck · Jan Fostier · Piet Demeester · Peter A. Vanrolleghem

Abstract The effectiveness of distributed execution of computationally intensive applications (jobs) largely depends on the quality of the applied scheduling approach. However, most of the existing non-trivial scheduling algorithms rely on prior knowledge or on prediction of application parameters, such as execution time, size of input and output, dependencies, etc., to assign applications to the available computational resources. A major issue is that these parameters are hard to determine in advance, especially if the end user does not possess an extensive history of previous application runs.

In this work we propose an online method for execution time prediction of applications, for which execution progress can be collected at run-time. Using dynamic

M. Chtepen (✉) · B. Dhoedt · F. De Turck · J. Fostier · P. Demeester
Department of Information Technology Broadband Communication Networks, IBCN, Ghent University, IBBT, Gaston Crommenlaan 8, Bus 201, 9050 Gent, Belgium
e-mail: maria.chtepen@intec.ugent.be

B. Dhoedt
e-mail: bart.dhoedt@intec.ugent.be

F. De Turck
e-mail: filip.deturck@intec.ugent.be

J. Fostier
e-mail: jan.fostier@intec.ugent.be

P. Demeester
e-mail: piet.demeester@intec.ugent.be

F.H.A. Claeys
MOSTforWATER N.V., Sint-Sebastiaanslaan 3a, 8500 Kortrijk, Belgium
e-mail: fc@mostforwater.com

P.A. Vanrolleghem
modelEAU, Département de génie civil et génie des eaux, Pavillon Adrien-Pouliot, Université Laval, 1065, avenue de la Médecine, Québec G1V 0A6, QC, Canada
e-mail: Peter.Vanrolleghem@gci.ulaval.ca

Published online: 17 March 2012

progress information, the total job execution time can be predicted using extrapolation. However, the predictions achieved by extrapolation are far from precise and often vary over time as a result of changing application dynamics and varying resource load. Therefore, to compute the actual job execution time we match a number of predefined *prediction evolution* models against the consecutive extrapolations, by adopting nonlinear curve-fitting. The "best-fit" coefficients allow for more accurate execution time prediction.

The predictions made are used to enhance a dynamic scheduling algorithm for workflows introduced in our earlier work. The scheduling algorithm is run with and without curve-fitting, showing a performance improvement of up to 15% in the former case.

Keywords Application distributed execution · Execution time prediction · Optimization

1 Introduction

When dealing with application complexity and long execution times, one often considers distributed solutions such as clusters and grids. However, the benefit of distributed approaches largely depends on the scheduling strategy applied. Most of the currently existing schedulers for distributed systems require sufficiently accurate information to be provided on the applications scheduled and on the available resources, to perform effective job–resource matchmaking. Unfortunately, this information is hard to obtain in advance, due to high job diversity, large variation in input parameters and the dynamic nature of distributed resources.

One of the major issues in the domain of distributed computing is considered to be prediction of application run-times. Taking into account the diversity of the existing applications, it seems extremely difficult to define a general solution for the problem. Therefore, in this work we concentrate on the category of applications for which the execution progress can be monitored at run-time. We propose a dynamic prediction mechanism that iteratively refines estimates of job execution time based on periodic run-time updates on the job progress.

Concretely, there exists a broad group of applications for which execution progress can either be collected periodically or at particular time-points during the application execution. Examples of such applications are simulations with a total simulated time T^{total} that is known *a priori* (see Table 1 for a listing of symbols). If, for instance, the current simulated time (t) can be collected at run-time, we can predict the total execution time of our job J (E_J^{est}) using extrapolation: $E_J^{\text{est}} = T_J * (T^{\text{total}}/t)$, where T_J is the processing time of J thus far. Other examples are applications consisting of a number of consecutive runs. When the total and the current number of runs are known, we can extrapolate as above, to predict E_J^{est}. The problem of this simple approach is that the predicted execution time E_J^{est} can strongly vary over time, depending on system dynamics, complexity of individual job runs and changes in resource load. To address this dynamic behavior of applications and distributed resources, we modify E_J^{est} at run-time, taking into account the previous job execution time predictions.

Table 1 Listing of frequently used acronyms and symbols

Parameter	Description
A	Amplitude of estimates' oscillations
b	Factor determining speed in increase/decrease of A
B	Network bandwidth
CR	Computational Resource
CR^{ref}	Reference Computational Resource
CS	Checkpoint Server
DAG	Directed Acyclic Graph
E_J^{act}	Actual execution time of task J
E_J^{CR}	Execution time estimation of task J on resource CR
E_J^{est}	Estimated execution time of task J
E^{ref}	Reference execution time used in progress models
F	Oscillation curve period
$Func$	Set of prediction evolution functions
GS	Grid Scheduler
$Hist$	Set of previous estimates and progress collection timestamps
i	Initial parameter value
$Init$	Set of initial parameter values for optimization
IS	Information Service
LAN	Local Area Network
LASP	Double Exponential Smoothing
MIPS	Million Instructions Per Second
$MIPS_{\text{CR}}$	Computational capacity of resource CR
$MIPS_J^{\text{CR}}$	Computational capacity of resource CR allocated to J
MPI	Message Passing Interface
n_{CR}	Number of tasks running on resource CR
$n_{\text{CR}}^{\max}$	Maximum number of tasks on CR
ψ	Prediction evolution function
P_J	Progress of task J
P_J^{perc}	Percentage of task J completed
PS	Parent Set
r_2	White noise coefficient
$ResNorm$	Squared 2-norm of the residual
S	Grid Site
t	Current simulated time
T_J	Processing time of task J thus far
T^{total}	Total simulated time of a simulation job
UI	User Interface
φ	Oscillation curve phase
WAN	Wide Area Network
X	Vector with best-fit parameter values for certain input data

We make the realistic assumption that E_J^{est} converges over time to a certain end point, which means that the longer a job is running, the closer E_J^{est} converges to the actual job execution time (E_J^{act}). Therefore, we can match the course of E_J^{est} against a number of *prediction evolution* models, determined using historical information on previous application runs. The matchmaking is realized using a curve-fitting optimization procedure. Parameters determined by the optimization provide an accurate estimate of the convergence point and thus of the total execution time.

Obviously, only a dynamic scheduling approach can benefit from the proposed prediction mechanism. A scheduler should be able to assign arriving jobs randomly to the available resources and to reschedule them at run-time as more information on individual job progress becomes available. Therefore, to evaluate the performance of our prediction method, the latter is incorporated into a dynamic scheduling algorithm for workflow applications that is introduced in our previous publication [2]. The algorithm was simulated in a grid simulation environment (DSiDE [3]), using realistic workload derived from a modeling and simulation tool for environmental systems (Tornado [5]).

The remainder of this paper is structured as follows: Sect. 2 introduces related work; Sect. 3 describes the job/task execution time prediction method proposed; the simulation scenario utilized for evaluation of the prediction method is discussed in Sect. 4; simulation results can be found in Sect. 5; and, finally, Sect. 6 concludes the paper.

2 Related work

Currently existing approaches for estimating execution times of jobs running in distributed environments can be subdivided into two main categories: application component performance modeling and historical prediction.

Application component performance modeling considers the number and the complexity of instructions executed for particular input parameters. A concrete example of this approach can be found in [13], where a job is run initially with several small-size input problems. The computational complexity for each run is determined as a function of the number of floating point operations performed and the memory access pattern. After the data collection phase, least square curve-fitting is applied on the collected data to make prediction for a specific input data set. The main disadvantage of this approach is that it operates at a fine-grained instruction level and is, therefore, only applicable to small, deterministic applications with a limited number of input parameter combinations. A slightly different mechanism is proposed in [9]. Here the execution time prediction is taken to a next level of abstraction, by identifying a number of primitive routines performed by a job. The total execution time prediction is derived from the job performance within the routines. Other application model-based prediction solutions are discussed in [20] and [17]. In [20] dynamic models of workload evolutions are designed to predict the execution time of non-deterministic bulk synchronous computations on multiprocessors. In [17] a modeling approach to estimating the execution time of long-running scientific applications is presented. The approach is based on the observation of resource usage behavior of a job and job profiling.

In general, it can be concluded that while exhaustive profiling within application performance modeling provides for very accurate estimates, correct application models are hard or sometimes even impossible to obtain. Furthermore, the approach yields insufficient insights on the impact of input data changes on application execution.

On the other hand, the historical prediction method that utilizes sets of past observations to predict execution times, seems to be more effective and more generally applicable. Therefore, it is frequently applied within recent research projects. For instance, in [10] the ScoPred performance predictor is discussed, which applies multiple linear regression using rough estimates of application execution time provided by the end user and historical application run data to predict the execution times. Other regression-based methods are described in [11] and [8]. Here regression models and filtering techniques are applied on a subset of previous application runs in order to discover the relationships between variables that affect the run-times of applications (e.g., application input, resource capacities). In [14] and [15], *similarity template* based approaches are proposed. A *similarity template* refers to a set of selected attributes. In [14] matchmaking of templates is supervised by an expert user, who is supposed to indicate the relevance of each attribute for a particular application. In [15], on the other hand, similarity distance calculations are performed on attributes in a predefined order. Hereby, the similarity calculation of the second most relevant attribute will occur only for those cases that get high similarity for the most relevant attribute.

The main disadvantage of the historical approach is that a large number of historical records must be stored before matchmaking of a job against the available records can provide a sufficiently accurate estimate. However, the more records are stored, the longer the matchmaking procedure takes.

The approach proposed in this work can be classified as high level application performance modeling, where the possible models of job execution progress evolution are provided by end users in advance. The advantage of our approach is that it requires a relatively limited number of *prediction evolution* models to deliver acceptable prediction accuracy.

3 Prediction algorithm description

The algorithm proposed is primarily designed for dynamic schedulers assigning jobs with *a priori* unknown execution times within dynamic distributed environments. The mechanism is implemented as an independent module that can be plugged into a scheduler. The idea is that the scheduler periodically consults the prediction algorithm to determine a new job execution time estimate, based on the earlier collected job progress history. The remainder of this chapter presents the algorithm pseudocode and gives explanation on consecutive steps.

The functionality of the proposed algorithm can be formalized as follows:

Input: *Job name*: J,
Set of (estimate, progress timestamp)-pairs: $Hist = \{(E_1, T_1), \ldots, (E_n, T_n)\}$,
Set of initial parameter values: $Init = \{i_1, \ldots, i_m\}$,

Set of prediction evolution functions: $Func = \{(\psi_1, \ldots, \psi_k\}$
Output: *Job execution time estimate*: E_J^{est}

1: **if** $Size(Hist) \equiv GetPreviousEstimatesNo(J)$ **then**
2: $E_J^{\text{est}} \Leftarrow E_n$
3: **else**
4: $SetPreviousEstimatesNo(J, Size(Hist))$
5: **if** $Size(Hist) \equiv 1$ **then**
6: $E_J^{\text{est}} \Leftarrow E_1$
7: **else**
8: **for all** $\psi_j \in Func$ **do**
9: $[X_{\psi_j}, ResNorm_{\psi_j}] \Leftarrow MatchCurve(\psi_j, Init, Hist)$
10: **end for**
11: $\psi_j \Leftarrow MinResNormGet(ResNorm)$
12: $E_J^{\text{est}} \Leftarrow CalculateLimit(\psi_j, X_{\psi_j})$
13: **end if**
14: **end if**

The input to the algorithm consists of the following parameters: the name of a job (J), for which a new estimate of the execution time is required; the prediction history *Hist*, containing pairs of execution time predictions E, computed by extrapolation of consecutive progress measurements, and progress collection timestamps T; and a set, *Init*, of initial parameter values (i). The initial parameters serve to initialize the optimization performed in the scope of curve-fitting. The parameters within *Init* are provided by end users, which are presumed to possess sufficient application knowledge to provide for the appropriate initial values. A good choice of the latter is highly important for the accuracy of the prediction mechanism, since it avoids the optimization ending in a local optimum. Also the set, *Func*, of functions, ψ, is provided to the algorithm. Each function ψ describes a possible evolution over time of job execution time estimates. The historical input-data (E) is matched against the available functions to provide a new estimate, E_J^{est}, which is the output of the algorithm.

Before proceeding with calculating E_J^{est}, the algorithm first checks whether new information on job progress has become available since the last algorithm run. If the latter is not the case, the previous value of E_J^{est} still applies (see lines 1–2). On the other hand, when the number of extrapolated estimates increases ($Size(Hist)$), this number is saved for the next run and the algorithm proceeds with computing the new E_J^{est} (see lines 3–14).

Obviously, it is assumed that the prediction algorithm is called only after at least one progress indication is collected. However, since no curve fitting can be performed for a single point, the algorithm simply returns the initial estimate value E_1 (see lines 5–6). When the set *Hist* contains multiple estimates, for each predefined function ψ the curve-fitting optimization procedure *MatchCurve* is called (see lines 7–9). The *MatchCurve* method takes as arguments a function ψ, the initial parameter values and the execution time estimate data points, together with the estimate timestamps. The outputs of the method are two vectors: X contains the parameter values that best-fit function $\psi_j(X, T)$ to the data *Hist*; and *ResNorm*, which represents the residual norm, used by the prediction algorithm to determine the function ψ_j that best fits the

provided input data. *ResNorm* is calculated as the squared 2-norm of the residuals (see Formula 1), which means the parameter depicts the squared difference between the optimized function and the input data. Clearly, the smaller the difference, the better the curves fit.

$$ResNorm_{\psi_j} = \sum_k (\psi_j(Init, T_k) - E_k)^2. \quad (1)$$

Finally, the limit of the function with the minimum *ResNorm* is calculated, which provides us with the a new execution time prediction (see lines 11–12). As was mentioned previously, we assume that the longer a job is running, the closer its execution time prediction gets to the real execution time value. This means that the function representing the execution time evolution converges over time to a certain limit-value. Since there are always only a limited number of functions provided, the limit expression can easily be determined analytically by an end user and provided to the algorithm together with the *prediction evolution* functions (*Func*). Afterwards, the predictions can be calculated by substituting the X-parameters into the limit expression of the ψ_j-function.

4 Simulation experiment description

To evaluate the performance of the proposed prediction algorithm, we integrate the latter into the dynamic workflow scheduler introduced in our previous work [2]. The scheduler is implemented in an existing grid simulation environment, called DSiDE [3], which allows for easy modeling and monitoring of dynamic resource and application behavior. To get an accurate indication on the overhead periodic prediction and rescheduling introduce in distributed environments, a realistic medium-sized grid model and a workload model deduced from a real-world application are considered.

In the remainder of this section, the workload and the grid models, together with the utilized dynamic scheduling approach are discussed in more details.

4.1 Workload model

The performance of the proposed algorithm is simulated using a workload model derived from Tornado [5], an existing application for modeling and virtual experimentation with complex environmental systems. Tornado is particularly interesting as a use case since it generates jobs with strongly varying properties in terms of job execution times, mutual dependencies, size of input/output data, etc.

In this work we consider Tornado jobs composed of tasks with input dependencies. It means that some tasks require inputs generated by other tasks, before they can proceed with their execution. This type of dependency can significantly benefit from distributed execution, compared, for instance, to MPI-based (Message Passing Interface) [7] dependencies, since it does not require extensive communication between tasks at run-time.

In general, a Tornado job with input dependent tasks can be represented as a DAG (Directed Acyclic Graph) of the form shown in Fig. 1. As depicted in the figure, each job contains a single *initial task*, which generates inputs for one or several *dependent*

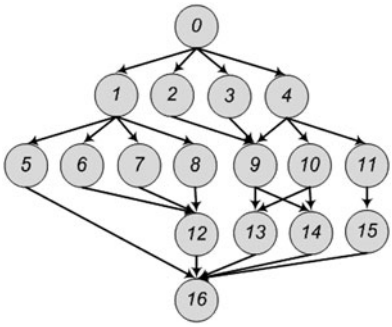

Fig. 1 Example of a workflow consisting of tasks with input-dependencies, organized into a DAG structure

tasks, which in turn generate inputs for their *dependent tasks*, etc. The procedure continues until the final level of the dependency hierarchy is reached, where a single *final task* produces job results. An example of such a dependency structure within Tornado is the Scenario Analysis experiment: the *initial task* determines different parameter values, input variable values and/or initial conditions; afterwards, individual simulation experiments (*dependent tasks*) are run with each combination of inputs and during each run the simulated trajectories of a number of selected quantities are saved; the *final task* is executed to compute a variety of objective values.

Due to a large diversity of possible inputs for Tornado experiments, it is hard to predict the execution time of an experiment in advance. However, for a large group of Tornado jobs the execution progress can be monitored at run-time and their total execution time can be predicted using extrapolation:

$$E_J^{\text{est}} = \frac{100\% \times T_J \times MIPS_{\text{CR}}}{P_J \times n_{\text{CR}}} \tag{2}$$

where J is a Tornado task running on a distributed computational resource CR; T_J is the wall clock execution time of J thus far; $MIPS_{\text{CR}}$ is the speed of the resource CR; P_J is the percentage of the task J completed within the time period T_J; and n_{CR} is the total number of tasks running on CR. In fact, we compute the execution time prediction on a theoretical reference resource CR^{ref}, having $MIPS_{\text{CR}^{\text{ref}}} = 1$ and $n_{\text{CR}^{\text{ref}}} = 1$.

To provide for a realistic model for evolution of execution time estimates, a number of Tornado experiments were observed. From these observations can be concluded that the estimate curves show strongly alternating evolution (see Fig. 2). However, a common tendency can be distilled by defining the following three approximation models:

- The *overestimate model* represents estimates that are gradually decreasing until the *stable state* is reached. In the *stable state* the exact execution time is known and the predictions no longer change. An example of this model is the "Galindo_CL" simulation experiment [4].
- The *underestimate model* is the opposite of the *overestimate model*. Here the execution time prediction increases until the *stable state*. "BSM1_CL" [4] is an example of this model.

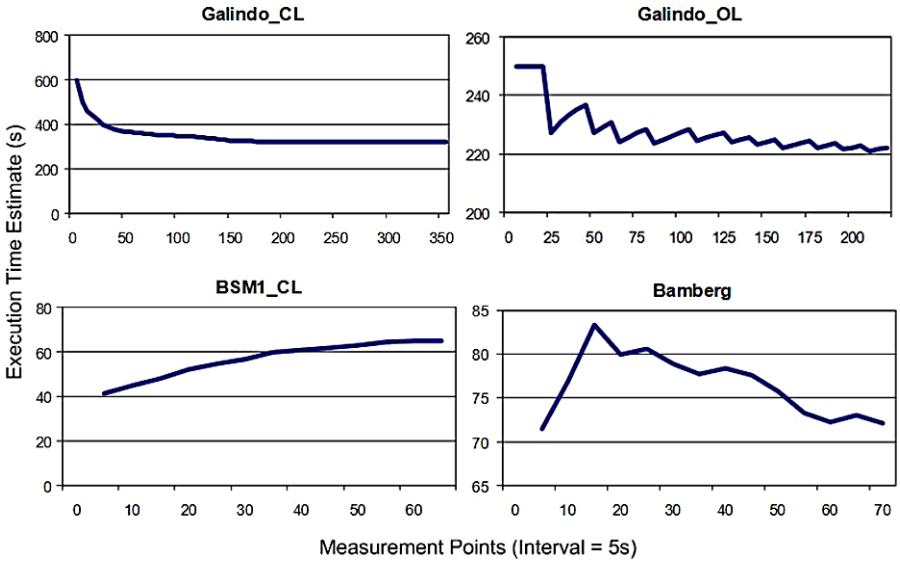

Fig. 2 Examples of evolution of execution time estimates for the "Galindo_CL," "Galindo_OL," "BSM1_CL" and "Bamberg" simulation experiments

– The *fluctuating model* represents an erratic pattern, whereby predictions oscillate over time. In the case of "Galindo_OL" [4] and "Bamberg" [4] we can talk about the *fluctuating model*.

The above-mentioned models can be approximated mathematically by the following exponential functions: $E_J^{\text{est}} = E_J^{\text{ref}} + r_1 A e^{-bt} \pm r_3, r_1 > 0$ describes the exponentially decreasing *overestimate model*; $E_J^{\text{est}} = E_J^{\text{ref}} - r_1 A e^{-bt} \pm r_3, r_1 > 0$ represents the exponentially increasing *underestimate model*; and, finally, $E_J^{\text{est}} = E_J^{\text{ref}} \pm r_1 A e^{-bt} \sin(2\pi F t + r_2\varphi) \pm r_3, r_1 > 0$ represents the *fluctuating model*. In these equations, E_J^{ref} is a reference execution time value for the job J that prevents the models from having too short initial execution time estimates; r_1 and r_2 are pseudo-random weight factors that can take values between 0 and 1; A is the amplitude of estimate oscillations; b is a weight factor that determines the speed in the increase/decrease of A over time; F stands for the oscillation curve period; φ represents the oscillation curve phase; and, finally, $r_3 = pA$ is white noise that is defined as a small percentage p of the amplitude. Obviously, the larger p, the more noise is imposed on the model and the more difficult it is to classify the curve. To eliminate/reduce noise, *filtering* or *smoothing* can be applied on the input data before the optimization step. Using these techniques, an approximate function can be constructed that captures the important patterns in the data and leaves small oscillations out.

Another issue is that it is often difficult to distinguish noise from the *oscillating model* pattern. However, we are not really interested in oscillations but rather in the end values, which remain after the oscillations have decayed. We partially address both issues by providing to the prediction algorithm the following two *prediction*

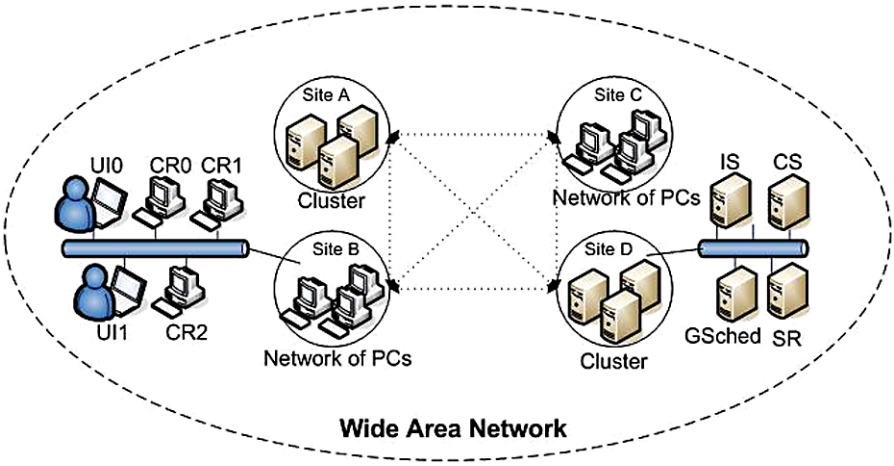

Fig. 3 Considered grid model: Computational Resource (*CR*), Grid Scheduler (*GS*), User Interface (*UI*), Information Service (*IS*), Checkpoint Server (*CS*)

evolution functions: $\psi_1 = x_1 + x_2 e^{(-x_3 t)}$ and $\psi_2 = y_1 - y_2 e^{(-y_3 t)}$, where x_j and y_j stand for the parameter values to be determined. To a certain extent, the functions have the effect of a *smoothing* technique, since after sufficiently long task run-times they capture the increasing/decreasing data pattern and manage to eliminate noise and oscillations. The limits of the functions can be analytically determined as x_1 and y_1.

4.2 Grid model

We model a grid environment (see Fig. 3) consisting of a number of distributed Sites (S), aggregating heterogeneous dedicated Computational Resources (CR).

In practice, resource capacity is a complex quantity, which is influenced by different hardware components. In this work, we use a simplified metric, called MIPS, which is often applied theoretically to compare resource capacities.

Another assumption relates the sharing of CR capacity among simultaneously active tasks. Normally, each application requires different and alternating amounts of hardware resources (CPU, IO bus, etc.), but we assume that each task J is allocated an equal share of a resource capacity ($MIPS_J^{\mathrm{CR}}$), which is determined using the following equation:

$$MIPS_J^{\mathrm{CR}} = \frac{MIPS_{\mathrm{CR}}}{n_{\mathrm{CR}}}. \tag{3}$$

To avoid overzealous partitioning of $MIPS_{\mathrm{CR}}$, the number of tasks allowed to run on a *CR* simultaneously is limited by the boundary $n_{\max}$: $n_{\mathrm{CR}} \leq n_{\max}$.

Next to CRs, the considered grid infrastructure includes a number of general services, such as a Grid Scheduler (GS), which maintains a job queue and is responsible for task-resource matchmaking; several distributed User Interfaces (UIs), utilized by end users to submit their applications to the grid; an Information Service (IS) required to collect information on the grid status; a Checkpoint Server (CS) where checkpoints

are saved; and a Storage Resource (SR) where output data is transferred after a job execution.

Often, a real-world grid infrastructure contains several SRs. The allocation of input/output data to an appropriate SR is then performed using a certain data scheduling policy. Since data scheduling is out of the scope of this work, we limit the considered grid infrastructure to a single SR.

The performance evaluation of the proposed prediction algorithm would not be accurate if we would not consider different types of overhead, related to the dynamic information collection and the rescheduling process.

The first important source of overhead relates to the grid middleware services: querying IS for dynamic resource/task status, periodic task execution time predictions, (re)schedule computations by GS, and, finally, checkpointing and migration slow down task execution. These types of overhead are taken into account by adding constant delays to our model.

The second significant cause of task slow down originates from network transfers, such as input/output file staging, workflow rescheduling and rollback. In our model all grid sites are interconnected by a Wide Area Network (WAN), while the communication within the sites go by different Local Area Networks (LANs). It is assumed that network transfers within LANs originate exclusively from grid jobs, while WANs are open to external network traffic. Therefore, two different models, described in [1], are used for bandwidth (B) sharing among simultaneous transfers: every data transfer route going through a WAN link gets a small equal share of the total link capacity; while capacities of LAN links are proportionally shared among simultaneous active grid transfers. For simplicity we assume that total link capacities do not change over time and that links are not subject to failure.

4.3 Dynamic scheduling algorithm

In this work we integrate our execution time prediction module into a dynamic scheduling algorithm, to observe to what extent the predictions made improve the algorithm performance. The algorithm, originally proposed in [2], operates in dynamic grid environments where tasks with input dependencies and unknown execution times (for which, however, periodic progress information can be collected) are run. The algorithm makes use of information services to collect dynamic system updates and applies these updates to (re)schedule dependent tasks. Figure 4 gives a brief overview of the different algorithm steps, which are described in more details in the remainder of this section.

The objective of the algorithm is to reduce the execution time of a job (see Fig. 1) running on a set of distributed heterogeneous resources, by taking into account task interdependencies. Before we proceed, we define the notion of a *parent set* (PS), which is a set of tasks generating input for the same group of *dependent tasks*. For example, in Fig. 1 tasks $\{0\}$, $\{1\}$, $\{2, 3, 4\}$, $\{4\}$, $\{6, 7, 8\}$, $\{9, 10\}$, $\{11\}$, $\{5, 12, 13, 14, 15\}$ form *parent sets* for respectively tasks 1–4, 5–8, 9, 10–11, 12, 13–14, 15 and 16. Clearly, each task in the considered workflow, except for the *initial task*, has a *parent set*. *Parent sets* of different tasks are not necessarily unique and they can overlap, which is the case for the sets $\{2, 3, 4\}$ and $\{4\}$ in the example above.

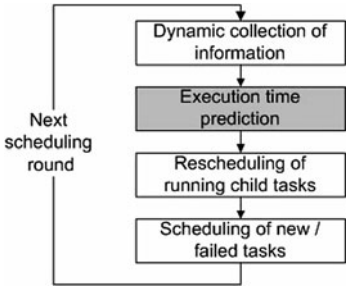

Fig. 4 Flow of the operation phases of the dynamic scheduling algorithm

Several issues arise when we are dealing with input-dependency constraints. First of all, if jobs arrive with high frequency into a grid, *initial tasks*, which do not have input dependencies and can be started immediately, occupy the available resources to a large extent. This leads to a delay in the execution of *dependent tasks* and thus prolongs the job execution as a whole. Secondly, not all tasks within a *parent set* are computationally equally intensive, which means that some tasks may finish much faster than others, when executed on resources with similar capacity. The imbalanced task execution, however, is not advantageous because the output of a whole *parent set* is required to proceed with the execution of its *dependent tasks*.

Since we are usually not interested in partial results but only in the results produced by *final tasks*, the algorithm tries to reduce the execution time of a job at the cost of possibly slower execution of individual tasks. The idea behind the algorithm is to give the processing of *dependent tasks* a higher priority than the execution of *initial tasks*. In fact, the latter are scheduled only when no *dependent tasks* are waiting for the execution. Furthermore, the execution of *parent sets* is balanced by scheduling the computationally most intensive tasks within a set to the fastest available resources, leaving slow resources to short tasks. Our optimization criteria can formally be defined as:

$$\begin{cases} \min\limits_{\forall PS}\Big(\max\limits_{\forall J_i \in PS}\{E_{J_i}\}\Big) \\ \forall PS\colon \min\limits_{\forall J_i, J_j \in PS} |E_{J_i} - E_{J_j}|. \end{cases} \tag{4}$$

The above equations mean that the maximum execution time ($\max_{\forall J_i \in PS}\{E_{J_i}\}$) as well as the difference of task execution times ($|E_{J_i} - E_{J_j}|, \forall J_i, J_j \in PS$) within each *parent set* should be minimized. Clearly, to satisfy these criteria and to provide an appropriate schedule, the algorithm requires a good task execution time estimate mechanism. The better the provided estimate, the less rescheduling needs to be performed on each system/task dynamic information update.

Concretely, the algorithm of Fig. 4 proceeds as follows:

- *Collection of dynamic information*. The information on resource load and availability, as well as the information on job status and progress of running jobs is collected. Important to mention is that the data collected can be outdated, depending on the length of the interval used by the IS to query the grid.
- *Execution time prediction*. In this step, the execution time predictions of tasks within *running PSs* (RPSs) are (re)computed, based on updates in task progress

and resource status. By the term *running PS* we understand a PS that exclusively contains finished tasks and tasks actually running on active grid resources, but not waiting tasks.
- *Rescheduling of Running PSs*. Tasks within RPSs are reassigned to balance their predicted execution times: the longest task is assigned to the fastest available resource (minimizing maximum execution time), while other tasks are assigned such that their execution times are as close as possible to the execution time of the longest task within the RPS, without actually exceeding it (minimizing processing time difference). This means that the shortest tasks are migrated to slowest resources, leaving the fastest resources to the tasks requiring fast execution.
- *Scheduling of idle tasks*. The *parent sets* containing idle tasks are assigned to the resources remaining after the rescheduling step. The PSs are processed in the order of their arrival into the GS-queue. For some applications we may possess an initial estimate of the total tasks execution time. In this case the scheduler proceeds as described in the previous step. Otherwise, tasks are assigned randomly.

  More detailed information on the different steps of the algorithm can be found in [2].

5 Algorithm performance evaluation

To measure the benefit of the proposed prediction algorithm for complex jobs with unknown execution times, a number of simulation experiments were performed in the DSiDE simulation environment. In this section we describe the simulation scenario parameters (see Table 2), together with the performance results.

5.1 Simulation experiment description

A grid consisting of 4 computational sites, with 32 CRs each, is observed during 24 hours of simulated time. Each CR within the grid has a computational capacity between 0.5 and 4 MIPS (uniformly distributed among CRs) and is limited to run a maximum of 2 tasks simultaneously. To focus on the variation of the execution time progress and on the accuracy of the predictions, we assume that once initialized, the computational speed of CRs remain unmodified during the whole simulation experiment.

The WAN links connecting the 4 distributed sites transfer data at a constant rate of 5 Mbit/s, with a latency uniformly distributed between 3 and 10 ms per link. On the other hand, intra-site transfers occur with a maximum speed of 1 Gbit/s. However, since link capacity within LANs is shared among the active data transfers, the more data traffic has to be processed, the lower the transfer rate. Finally, all the LAN links possess a fixed latency of 1 ms per link within a transfer route.

As was mentioned earlier, job parameters for the simulated grid model are derived from an existing tool for modeling and virtual experimentation with environmental systems, called Tornado. Tornado possesses a broad category of jobs, or *virtual experiments*, with input dependencies. In this work, we consider a group of input dependencies resulting from *model sweeps*. It means that tasks are derived from different mathematical models and have strongly varying execution times. The considered execution times (E^{act}) on CR^{ref} and their variations are depicted in Fig. 5.

Table 2 Listing of model parameters

Parameter	Value
A	$U(E^{\text{ref}} \times 150\%, E^{\text{ref}} \times 200\%)$
Activation interval of IS	5 min
b	$U(0, 1)$
B of *LAN* links	1 Gbit/s
B of *WAN* links	5 Mbit/s
C	2 s
C_J	1 GB
CR number	128
F	$U(E^{\text{ref}} \times 5\%, E^{\text{ref}} \times 10\%)$
I_J	1 GB
Latency of *LAN* links	1 ms
Latency of *WAN* links	U(3 ms, 10 ms) ms
$MIPS_{\text{CR}}$	U(0.5 *MIPS*, 4 *MIPS*)
$MIPS_{\text{CR}^{\text{ref}}}$	1 *MIPS*
$n_{\text{CR}}^{\max}$	2
$n_{\text{CR}^{\text{ref}}}^{\max}$	1
O_J	1 GB
P	U(1%, 100%)
Scheduling interval of GS	10 min
φ	$U(E^{\text{ref}} \times 1\%, E^{\text{ref}} \times 2\%)$

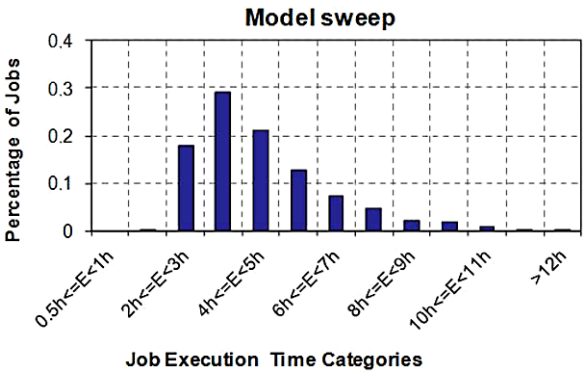

Fig. 5 Task length distribution for model sweep-based jobs

In this work, we simulate jobs containing 10 dependent tasks on average, organized into a 3-level dependency hierarchy. For simplicity, we assume that input, output, and checkpointing data of all tasks are equally sized and amount to 1 GB. A checkpointing delay of 2 s is also identical.

The task parameters considered correspond with the actual task properties observed when running Tornado experiments on the UGent grid infrastructure [18]. The UGent grid is a part of the Belgian grid infrastructure and consists of 76 CRs having a total of 222 CPUs, 304 GB memory and 4.4 TB disk space.

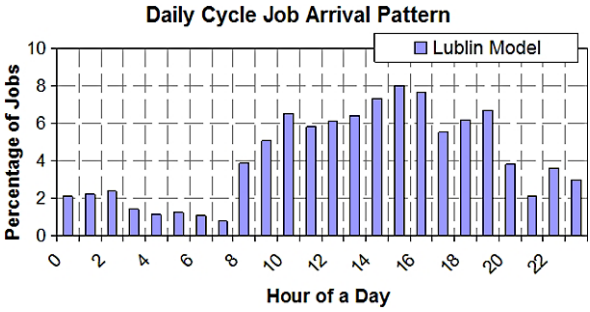

Fig. 6 Job arrival pattern with daily cycle. $U(U_{\min}, U_{\max}) =$ uniform distribution within $U_{\min}$ and $U_{\max}$

The workload described above is submitted into the considered grid environment according to the Lublin job generation model [12]. The model implies that most of the jobs (about 80%) arrive during day time, between 8 AM and 8 PM, resulting in peak hour loads alternating with relatively idle periods, as shown in Fig. 6. This cyclic behavior largely corresponds to the behavior of Tornado users observed on the UGent grid.

We assume that an equal number of tasks following the *overestimate*, the *underestimate* and the *fluctuating* progress evolution models are submitted. The model parameters are initialized as follows (see also Table 2): E^{ref} for the *overestimate* model is uniformly distributed between $E^{\text{act}} \times 150\%$ and $E^{\text{act}} \times 200\%$; E^{ref} for the *underestimate* model equals $E^{\text{act}} \times 10\% + A$; E^{ref} for the *fluctuating* model equals E^{act}; A is uniformly distributed between $E^{\text{act}} \times 150\%$ and $E^{\text{act}} \times 200\%$; b is uniformly distributed between 0 and 1; F is uniformly distributed between $E^{\text{act}} \times 5\%$ and $E^{\text{act}} \times 10\%$; φ is uniformly distributed between $E^{\text{act}} \times 1\%$ and $E^{\text{act}} \times 2\%$; and, finally, we observe 3 types of noise $r_2 = 0$ (no noise), $r_2 = 0.05A$ (low noise oscillation amplitude) and $r_2 = 0.5A$ (high noise oscillation amplitude). The task execution progress is updated every $E^{\text{act}} \times 0.5\%$.

Finally, it is also important to mention that we utilized the Tornado modeling and virtual experimentation framework for fitting periodic extrapolated execution time predictions to a number of predefined functions within the prediction algorithm. Tornado routines were called from the DSiDE code using the TornadoCPP Software Development Kit (SDK) that consists of a Dynamically Linked Library (DLL), an import library and corresponding header files. In particular, to perform nonlinear curve-fitting (data-fitting) within Tornado, the following procedure is to be followed:

- Implementation of the prediction evolution functions as algebraic models, specified in one of the two modeling languages supported by Tornado: Model Specification Language (MSL) [19] or Modelica [6].
- Creation of a Simulation Experiment (ExpSimul) that simulates the models over the desired time interval.
- Creation an Objective Evaluation Experiment (ExpObjEval) that calculates the Sum of Squared Errors (SSE) between the values simulated by ExpSimul and the input data.
- Creation of an Optimization Experiment (ExpOptim) that executes ExpObjEval iteratively for different model parameters, until the minimum SSE is reached. We

have used the Simplex [16] optimization solver to provide initial model parameter values.

5.2 Simulation results

In this section we compare the performance of the dynamic algorithm when using two different task execution time prediction approaches. In the first approach the prediction value is simply derived by extrapolation from the last task progress measurement. The second approach is the proposed curve-fitting-based prediction mechanism. The performance of both methods is observed for job progress curves with different amplitude white noise. An example for the *overestimate* model in Fig. 7 suggests that we consider job execution time predictions with a perfectly exponential course (or a sinusoidal course in the case of the *fluctuating* model), as well as jobs with noise with low and high amplitudes.

The simulation results on the performance of the two prediction algorithms are depicted in Fig. 8. Concretely, Fig. 8(a) shows the mean fraction of *useful workload* processed by the dynamic scheduler in both cases. The term *useful workload* refers to the total processing time on CR^{ref}, spent running successfully executed jobs. Formally this definition can be written as follows:

$$E = \frac{\sum_{J \in Done} E_J^{\text{CR}^{\text{ref}}}}{\sum_{J \in Submitted} E_J^{\text{CR}^{\text{ref}}}} \tag{5}$$

where *Done* is a set of jobs, for which the final result is successfully computed within the observed time interval; and *Submitted* is a set of all submitted jobs. We have to emphasize that successfully executed tasks, belonging to jobs that have not managed to execute within the predefined interval, do not contribute to the *useful workload*.

The simulation results suggest that when use is made of the curve-fitting-based prediction method, the dynamic algorithm achieves up to 15% better performance (in the optimal case of "Low Noise"), compared to the extrapolation method. The advantage of the curve-fitting procedure is particularly remarkable in the case of noise with low amplitude ("Low Noise"). In this case the trend of the progress curve is preserved, simplifying the selection of an appropriate *prediction evolution* function and thus giving a correct indication of the total execution time in an early stage of a task processing. It means that the task can be assigned to the best suited resource at the beginning of its execution. Furthermore, matchmaking with an exponential curve largely eliminates oscillations in prediction values, reducing overzealous checkpointing and migration. The latter statement is confirmed by the simulation results in Figs. 8(b)–8(d), which shows respectively the proportions of rescheduling operations, network traffic and checkpoints performed by the curve-fitting-based and the extrapolation-based algorithms.

Obviously, when the prediction curves show smooth evolution ("No Noise"), extrapolation-derived values show less variation and thus need less rescheduling operations to calibrate task execution times. Therefore, the performance of the dynamic algorithm with extrapolation gets closer to the performance of the curve-fitting-based algorithm, compared to the "Low Noise" case.

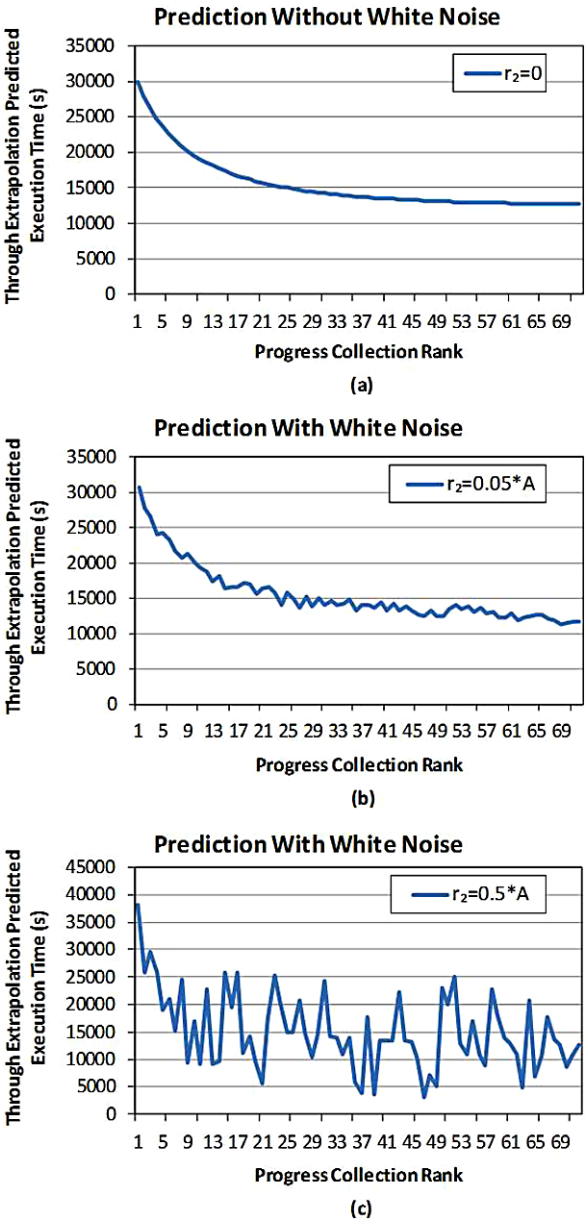

Fig. 7 Examples of white noise, used to evaluate the prediction algorithm performance: (**a**) exponential progress evolution without noise, (**b**) exponential progress evolution with low amplitude noise, and (**c**) exponential progress evolution with high amplitude noise

Finally, large amplitude white noise ("High Noise") conceal the trend of the prediction curve, obstructing the curve matchmaking procedure. The curve-fitting algorithm takes in this case wrong fitting decisions regularly, by selecting an inappropriate *prediction evolution* function to match with. In Figs. 8(b)–8(d) it can be observed that the numbers of checkpoints and migrations performed by both algorithms get close

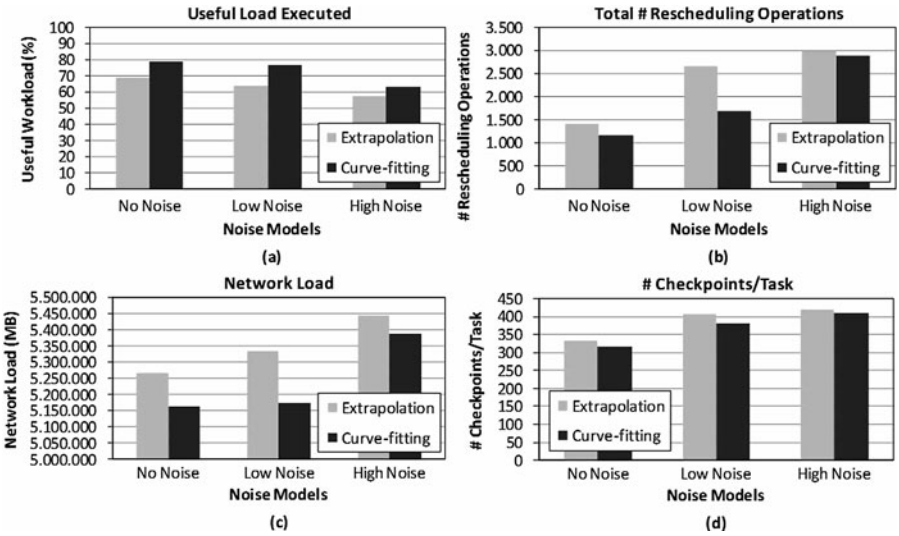

Fig. 8 Performance comparison between extrapolation-based and curve-fitting-based prediction approaches: (**a**) proportion of successfully processed *useful workload*; (**b**) network load comparison; (**c**) rescheduling proportion comparison; (**d**) checkpointing proportion comparison

to each other. As the result, both algorithms start performing similarly with respect to the *useful workload* processed.

6 Conclusion

In large distributed environments it is difficult to determine the execution time of applications due to a variety of input parameters, internal application dynamics and changing properties of distributed resources. However, knowledge of the total job execution time is essential for the implementation of an efficient scheduling policy. As this issue is difficult to address in a generic way, we consider a group of applications for which the execution progress can be monitored at run-time. An online prediction approach is proposed that uses the progress history to determine the course of the prediction curve and thus to estimate the total execution time. To achieve this goal, the approach makes use of curve-fitting of the current prediction evolution data to a number of predefined models.

To evaluate the prediction approach performance, the latter is integrated into an existing dynamic scheduler for workflow applications. The scheduler is in turn implemented in a grid simulator, called DSiDE, where a realistic medium-sized distributed environment with computational load derived from a real-world biological application is simulated. Under these conditions, the performance of the dynamic scheduler was evaluated for two situations: the scheduler uses the proposed prediction mechanism; the scheduler uses an extrapolation-based execution time predictor. The simulation results suggest the performance improvement of up to 15% in the former case.

References

1. Casanova H, Marchal L (2002) A network model for simulation of grid application. Tech rep, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme
2. Chtepen M, Claeys F, Dhoedt B, De Turck F, Demeester P, Vanrolleghem P (2009) Scheduling of dynamic workflows in grids. IEEE Trans Parallel Distrib Syst 20(2):180–190
3. Chtepen M, Claeys F, Dhoedt B, De Turck F, Vanrolleghem P, Demeester P (2006) Dynamic scheduling of computationally intensive applications on unreliable infrastructures. In: Proceedings of the 2nd European modeling and simulation symposium (EMSS '06), Barcelona, Spain
4. Claeys F (2008) A generic software framework for modelling and virtual experimentation with complex environmental systems. Phd thesis, Dept of Applied Mathematics and Process Control, Ghent University, Belgium
5. Claeys F, De Pauw D, Benedetti L, Nopens I, Vanrolleghem P (2006) Tornado: a versatile and efficient modelling & virtual experimentation kernel for water quality systems applications. In: Proceedings of the international environmental modelling and software conference (iEMSs), Burlington, Vermont, USA
6. Consortium, M.: Modelica website. http://www.modelica.org/
7. Gropp W, Lusk E, Doss N, Skjellum A (1996) High-performance, portable implementation of the MPI message passing interface standard. Parallel Comput 22(6):789–828
8. Iverson M, Ozguner F, Follen G (1996) Run-time statistical estimation of task execution times for heterogeneous distributed computing. In: Proceedings of the 5th IEEE international symposium on high performance distributed computing (HPDC-5 '96), Syracuse, New York
9. Iverson M, Ozguner F, Potter L (1999) Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. IEEE Trans Comput 48(12):1374–1379
10. Lafreniere B, Sodan A (2005) ScoPred—scalable user-directed performance prediction using complexity modeling and historical data. In: Proceedings of the 11th workshop on job scheduling strategies for parallel processing, Cambridge, MA, pp 62–90
11. Lee BD, Schopf J (2003) Run-time prediction of parallel applications on shared environments. In: Proceedings of the 5th IEEE international conference on cluster computing (CLUSTER'03), Hong Kong
12. Lublin U, Feitelson D (2003) The workload on parallel supercomputers: modeling the characteristics of rigid jobs. J Parallel Distrib Comput 63(11):1105–1122
13. Mandal A, Kennedy K, Koelbel C, Marin G, Mellor-Crummey J, Liu B, Johnsson L (2005) Scheduling strategies for mapping application workflows onto the grid. In: Proceedings of the 14th IEEE international symposium on high performance distributed computing, Research Triangle Park, NC, USA, pp 125–134
14. Nadeem F, Fahringer T (2009) Using templates to predict execution time of scientific workflow applications in the grid. In: Proceedings of the 9th IEEE/ACM international symposium on cluster computing and the grid, Shanghai, China, pp 316–323
15. Nassif L, Nogueira J, Ahmed M, Karmouch A, Impey R, Vinicius de Andrade F (2005) Job completion prediction in grid using distributed case-based reasoning. In: Proceedings of the 14th IEEE international workshops on enabling technologies: infrastructure for collaborative enterprise (WETICE), Linkoping, Sweden
16. Press W, Teukolsky S, Vettering W, Flannery BP (1992) Numerical recipes in C, Chap 10, 2nd edn. Cambridge University Press, Cambridge, pp 408–412
17. Sadjadi S, Shimizu S, Figueroa J, Rangaswami R, Delgado J, Duran H, Collazo X (2008) A modeling approach for estimating execution time of long-running scientific applications. In: Proceedings of the 22nd IEEE international parallel and distributed processing symposium (IPDPS), the fifth high-performance grid computing workshop (HPGC-2008), Miami, FL, USA
18. UGent: BeGrid UGent. http://begrid.atlantis.ugent.be/
19. Vanhooren H, Meirlaen J, Amerlinck Y, Claeys F, Vangheluwe H, Vanrolleghem P (2003) WEST: modelling biological wastewater treatment. J Hydroinf 5(1):27–50
20. Xu CZ, Wang L, Fong NT (2002) Stochastic prediction of execution time for dynamic bulk synchronous computations. J Supercomput 21(1):91–103