Diet-ethic: Fair Scheduling of Optional Computations in GridRPC Middleware

Frédéric Camillo, Eddy Caron, Ronan Guivarch, Aurélie Hurault, Cristian Klein, Christian Pérez

DIET-ethic: Fair Scheduling of Optional Computations in GridRPC Middleware

Frédéric CAMILLO[b], Eddy CARON[a], Ronan GUIVARCH[b], Aurélie HURAULT[b], Cristian KLEIN[a], Christian PÉREZ[a]

[a]LIP, UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668,
University of Lyon, France, `firstname.lastname@inria.fr`

[b]University of Toulouse - INPT (ENSEEIHT) - IRIT, `firstname.lastname@enseeiht.fr`

RESEARCH REPORT

N° 7959

10 Mai 2012

Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--7959--FR+ENG

Diet-ethic: Fair Scheduling of Optional Computations in GridRPC Middleware

Frédéric CAMILLO[b], Eddy CARON[a], Ronan GUIVARCH[b], Aurélie HURAULT[b], Cristian KLEIN[a], Christian PÉREZ[a]

[a]LIP, UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668, University of Lyon, France, firstname.lastname@inria.fr
[b]University of Toulouse - INPT (ENSEEIHT) - IRIT, firstname.lastname@enseeiht.fr

Project-Team Avalon

Research Report n° 7959 — 10 Mai 2012 — 16 pages

Abstract: Most HPC platforms require users to submit a pre-determined number of computation requests (also called jobs). Unfortunately, this is cumbersome when some of the computations are optional, i.e., they are not critical, but their completion would improve results. For example, given a deadline, the number of requests to submit for a Monte Carlo experiment is difficult to choose. The more requests are completed, the better the results are, however, submitting too many might overload the platform. Conversely, submitting too few requests may leave resources unused and misses an opportunity to improve the results.
This paper introduces and solves the problem of scheduling optional computations. An architecture which auto-tunes the number of requests is proposed, then implemented in the DIET GridRPC middleware. Real-life experiments show that several metrics are improved, such as user satisfaction, fairness and the number of completed requests. Moreover, the solution is shown to be scalable.

Key-words: HPC; GridRPC; malleable applications; Grid'5000

Diet-ethic: Ordonnancement équitable des calculs optionnels dans un intergiciel GridRPC

Résumé : La plupart des plate-formes HPC demandent à l'utilisateur de soumettre un nombre pré-déterminé de requêtes de calcul (aussi appelées « job »). Malheureusement, cela n'est pas pertinent quand une partie des calculs est optionnelle, c'est-à-dire, que l'exécution des requêtes n'est pas critique pour l'utilisateur, mais que leur complétion pourrait améliorer les résultats. Par exemple, étant donnée une date limite, le nombre de requêtes à soumettre pour une expérience Monte Carlo est difficile à choisir. Plus il y a des requêtes qui sont exécutées, meilleures sont les résultats. Cependant, en soumettant trop de requêtes, on risque de surcharger la plate-forme. À l'opposé, en ne soumettant pas assez de requêtes, les ressources sont sous-exploitées alors qu'elles auraient pu être utilisées pour améliorer les résultats.

Cet article introduit et résout le problème d'ordonnancer des requêtes optionnelles. Une architecture qui choisit automatiquement le nombre de requêtes est proposée puis implémentée dans l'intergiciel GridRPC DIET. Les expériences faites sur de vraies plate-formes – telles que Grid'5000 – montrent que plusieurs métriques peuvent être améliorées, telles que la satisfaction des utilisateurs, l'équité et le nombre des requêtes exécutées. Enfin, la solution proposée passe à l'échelle.

Mots-clés : HPC; GridRPC; applications malleables; Grid'5000

1 Introduction

High-Performance Computing (HPC) resources, such as super-computers and clusters, are managed by resource management systems, such as batch schedulers [1]. In order for a user to do computations on such resources, she has to submit a predetermined number of *requests* (also called jobs). For example, in order to execute a Parameter-Sweep Application (PSA), users generally submit one request for each of the parameters that are to be explored. Otherwise, they can coalesce multiple parameters in a single request, as done using the pilot job abstraction [2]. After all requests have completed, the user gathers the results and analyses them.

Unfortunately, choosing the requests to submit is cumbersome to do in advance for applications which have **optional computations**, i.e., computations that are not critical to the user, but their completion would improve results. For example, a widely used method to test the numerical stability of complex simulations is *sampling-based uncertainty analysis* [3], such as *Monte Carlo* experiments. Applications range from aerospace engineering to validating nuclear power plant design. At its core, the method consists in varying input parameters and studying the changes in the output parameters. The larger the number of tested input parameters, the better the quality of the results are. Hence, a typical user would like to test as many parameters as possible before a given deadline.

To run such computation on an HPC platform, the user would have to choose a number of requests to submit. If too many requests are submitted, then the resources are over-utilized, preventing *other* scientists from completing their simulations in due time. If too few requests are submitted, resources might be left idle, thus, the user lost an opportunity to improve her results. Hence, the user faces the difficulty of choosing the number of requests to submit.

Given the current platforms, finding a solution to this problem is difficult. An attempt at solving it has been done in the context of Cloud computing. Amazon proposes spot instances [4], which are virtual machine instances that can be terminated whenever the Cloud manager chooses to. Spot instances can be used to contain optional computations, as opposed to HPC instances [5] which contain mandatory computations. But even such an approach does not guarantee fairness. Intuitively, there is no guarantee that the system balances resources among the optional computations of each user.

This paper introduces and solves a new scheduling problem: fair scheduling of optional computations. Our contribution is three-fold: first, we present a motivating example and formulate the problem statement of this novel scheduling problem; second, we present a platform architecture which efficiently solves this problem; third, we evaluate our approach and show through real experiments its feasibility using a production-level GridRPC [6] middleware called DIET [7]. Results show that **user unhappiness** can be reduced to 0 and that **unfairness** can be decreased by up to 150 times.

The remaining of the paper is organized as follows: Section 2 presents a motivating use-case, which is then formalized into a problem statement in Section 3. Section 4 describes an architecture which solves the stated problem, and which is evaluated in Section 5. Section 6 compares our approach with related work. Section 7 concludes the paper.

2 A Motivating Use-case

This section briefly presents the GRID-TLSE project and a motivating use-case called multiple threshold pivoting.

The main goal of the GRID-TLSE project [8] is to design an expert site that provides easy access to many direct solvers for sparse linear systems, allowing their comparative analysis. The site assists users in choosing the right solver for their problems, appropriate values for the control parameters of the selected solver and the best computer architecture. GRID-TLSE also serves as a testbed for experts in sparse linear algebra. A computational Grid is used to execute all the runs arising from user requests, which is accessed using GridRPC. The GRID-TLSE project has started in 2002 years and is currently being used by 157 users.

Let us detail how direct solvers work. Solving a linear system of the form $Ax = b$ usually consists of three steps: analyse, factorisation and solve.

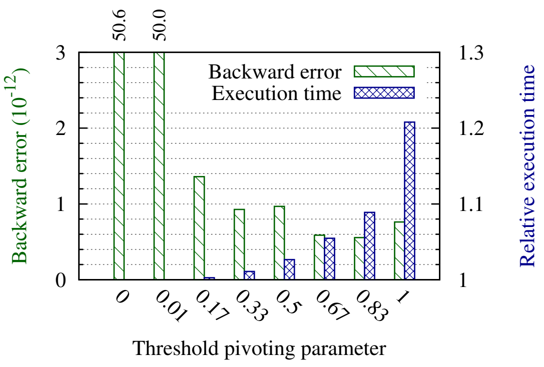

Figure 1: Example of the results output by a direct solver; backward error is defined as $||b - Ax||/||b||$

During the factorisation step of an asymmetric matrix, the matrices L and U are computed such that $LU = A$. In order to preserve the numerical stability (e.g., avoiding the division by a small number) pivoting can take place. One way to select the pivot is to choose a diagonal entry according to a given threshold. The selection of the threshold is important and for some values, the result of the solution of the linear system can be very bad (see Figure 1).

An often requested feature is to allow a user to run as many factorizations as possible, with different thresholds, until a given deadline is reached. If the deadline is too tight or resources are highly loaded, the system should test at least 3 thresholds, for example. In other words, the user needs to submit a few **mandatory requests**, which need to be solved whether the deadline is due or not, and a large number of **optional requests**, which the user would like to have computed, but are not as useful as to wait for their completion past the deadline.

Using GridRPC, it is difficult to choose the number of optional requests to submit to the platform. Obviously, if too few requests are submitted, the number of tested thresholds is suboptimal. If too many are submitted, the optional requests of other users might not have a chance at getting executing, which would be **unfair**. Even worse, the platform might be so overloaded that the other users might need to wait past their deadlines for the completion of the mandatory requests, which makes them **unhappy**.

3 Problem Statement

This section formalizes the problem. First, the resource and user models are described. Second, the metrics that the system has to optimize are defined.

3.1 Resource Model

Let the platform be composed of n_R resources, which are *homogeneous* (a computation request has the same execution time on any resource), *static* (resources are neither added nor removed during execution) and *reliable* (resources do not fail).

This model is somewhat simple but still applicable in many cases. For example, it fairly well approximates production-level multi-cluster systems such as the Decrypthon grid [9]. Nevertheless, these assumptions will be relaxed in future work.

3.2 User/Application Model

Let the platform be used by n_U users. A user i enters the system at time $t_0^{(i)}$ (which is not known in advance) and needs to solve at least $n_{min}^{(i)}$ (also called **mandatory** requests) and at most $n_{max}^{(i)}$ requests (including mandatory and **optional** requests).

The user sets a "tentative" **deadline** $d^{(i)}$ which acts as follows. If at time $d^{(i)}$ all mandatory requests are completed, the remaining optional requests are cancelled and the user exits the system. Otherwise, the user waits until all mandatory requests are completed, even if this means waiting past the deadline. In the latter case, optional computations can still be executed until the last mandatory request finishes. In other words, the hard deadline is equal to the maximum between the user-provided deadline and the last completion time of the mandatory requests.

To completely characterize the workload, the execution times need to be modeled. We consider that the requests of user i are homogeneous, having the same **execution time** $T^{(i)}$. This is a reasonable approximation for the targeted use-cases (Figure 1), as well as many parameter-sweep applications [10]. However, execution times are not known in advance.

3.3 Metrics

To evaluate how well a system deals with a workload, the following metrics are of interest: the number of unhappy users, unfairness and the number of completed requests.

The **number of unhappy users** is the number of users who did not complete their mandatory requests before their deadline $d^{(i)}$. These users had to wait additionally, after the tentative, user-provided deadline. Ideally, the number of unhappy users should be 0, provided the workload permits such a solution.

Before defining fairness, let us introduce some helper notations. For each user i, the amount of *deserved* resources $r^{(i)}_{deserved}$ (i.e., the amount the system *should* allocate the user) is computed as follows. The time is divided into time-slots, such that, during each time-slot j of start-time $S^{(j)}$ and end-time $E^{(j)}$, the set of users inside the system $U^{(j)}$ is constant:

$$U^{(j)} = \left\{ i : \left(t_0^{(i)} \le S^{(j)} \right) \wedge \left(E^{(j)} \le d^{(i)} \right) \right\}$$

The resource area (number of resources times duration) available during that time-slot is divided equally among the among users in $U^{(j)}$:

$$r^{(i)}_{deserved} = \sum_{j : i \in U^{(j)}} \frac{n_R \cdot \left(E^{(j)} - S^{(j)} \right)}{\# U^{(j)}}$$

Next, for each user i, the satisfaction $s^{(i)}$ is defined as the amount of resources the system allocated her $r^{(i)}_{allocated}$ over the amount of resource she deserved $r^{(i)}_{deserved}$. A satisfaction $0 \le s^{(i)} < 1$ means that the user i was allocated fewer resources than deserved, while $s^{(i)} > 1$ means that the user i was allocated more resources than deserved. Ideally, the satisfaction of all users should be 1, i.e., they are allocated as many resources as deserved.

Having all prerequisites, let **unfairness** be defined as the difference between the maximum and the minimum among the user satisfactions:

$$\text{unfairness} = \max_i s^{(i)} - \min_i s^{(i)}$$

We refrained from defining unfairness as an average, for the same reasons as stated in [11]: users tend to be more sensitive to fairness than to performance, thus, the former should be more like a law or a guarantee that the system is heavily penalized for breaking. Ideally, unfairness should equal 0.

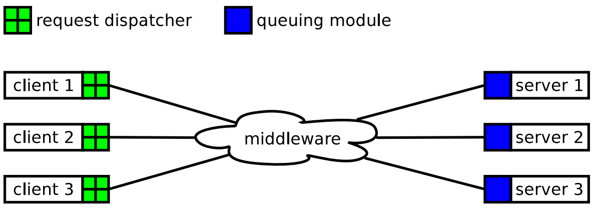

Figure 2: DIET-ethic architecture

Finally, the **number of completed requests** is a performance-oriented metric, computed as the sum of all the requests (mandatory and optional) belonging to any user that have completed.

To sum up, we aim at finding a system, which minimizes the number of unhappy users, minimizes unfairness and maximizes the number of completed requests, in this order. Note that, the three presented metrics can only be computed a posteriori, after all users exited the system.

4 Diet-ethic

In this section, the DIET-ethic platform for fair scheduling of optional computations is presented. First, the architecture is described in an abstract, implementation-independent manner. Second, our implementation of DIET-ethic in the production-level DIET GridRPC middleware is detailed.

4.1 Diet-ethic Extension

DIET-ethic is an extension over a client-server architecture. The clients (representing the users of the system) are resource consumers that generate computational requests, while the servers are resource providers, doing computations on behalf of the clients. Clients and servers are connected through a middleware, that implements a discovery mechanism.

Before describing DIET-ethic, let us highlight some design choices. First, we chose to keep the server-side scheduling algorithm simple and make servers unaware of the deadlines of the users. This choice has been taken so that clients, which are under the control of the user, be able to evolve their scheduling algorithms separately from the functionality offered by the server. For example, a future version of DIET-ethic might schedule workflows containing optional computations, without having to change the servers.

Second, the number of requests stored in the platform have to be minimized. This is important, since, for the targeted use-cases, any user could alone fill the whole platform with her optional requests. Therefore, having a system in which all users submit all their requests to the system would clearly not scale.

Let us now describe the DIET-ethic architecture consisting of a server-side queuing module and a client-side request dispatcher (Figure 2), that are described in the next sections.

4.1.1 Server-side Queuing Module

When a request arrives at a server, it is not immediately solved, but is added to a local queue. The queue is regularly checked to determine which requests should be started, cancelled (erased from the queue) or killed (prematurely terminated after it has been started). The queued requests are ordered as follows:

1. mandatory requests have priority over optional requests; this ensures that the number of unhappy users is minimized;
2. started requests have priority; otherwise resources might be wasted as requests are killed and computations completed so-far are lost. In the end, this improves the number of completed requests, paying a small price on fairness;
3. mandatory requests are order by submit time, i.e., the First-Come-First-Serve (FCFS) scheduling strategy is used; this allows users who arrived first in the system to get a better chance at completing their mandatory requests before the deadline, thus decreasing the number of unhappy users;
4. requests of users who had the fewest allocated resources so far have priority: this ensures server-local fairness;
5. for users having the same amount of allocated resources (such is the case when users enter the system as the same time) a request is randomly chosen. This ensures global fairness, as each server most likely choses to execute the request of a different client.

If a request gets to the front of the queue it is started. Otherwise, if sorting moves a request from the front of the queue, it is killed. For example, if a user submits a mandatory request to a server which is currently executing an optional request, the latter is killed. Requests can be cancelled on the client's demand.

4.1.2 Client-side Request Dispatcher

On the client-side, a custom request dispatcher is required. It works in three phases: setup, monitoring and cleanup.

The **setup phase** starts with a resource discovery, asking the middleware to return at most $n_{max}^{(i)}$ servers. Next, mandatory requests are dispatched to discovered servers–currently in round-robin. Finally, one optional request is submitted to each discovered server.

In the **monitoring phase**, the client enters an event-loop. It stays in this phase until all mandatory requests are completed and the deadline has not expired. When an optional request is completed, the client submits a new optional request to the server which executed the former request. As a result, as long as the user is in the system, provided $n_{max}^{(i)}$ is large enough, each server has at least one optional request in the queue, ready to be executed.

Finally, in the **cleanup phase**, the client cancels all requests that have been submitted but not yet completed. Note that, this phase is entered when all mandatory requests have been completed, thus, only optional requests need not be cancelled.

Having describe the architecture, let us pass on to implementing it in an existing GridRPC middleware.

4.2 Diet Middleware Architecture

Let us first describe the DIET GridRPC middleware architecture, which we have extended for optional computation support.

To simplify access to HPC resources, various programming models have been proposed. One of them is the GridRPC API [6], which has been standardized inside the Open Grid Forum (OGF) [12]. At its core, it extends the familiar Remote Procedure Call (RPC) paradigm to Grid environments. Hence, the end-users' client-server applications can

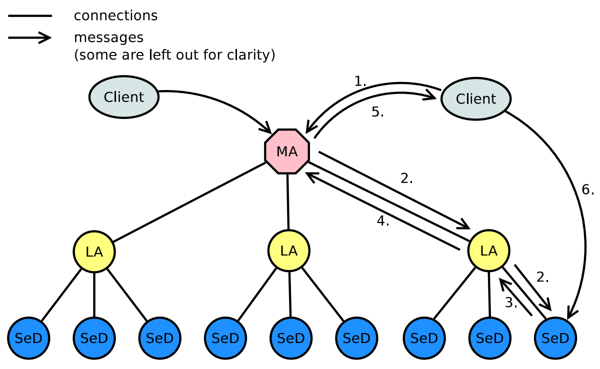

Figure 3: DIET architecture

be easily ported to Grids, with minimal effort. GridRPC offers mechanisms to make synchronous and asynchronous calls to a service. For the asynchronous case, clients can wait in a blocking or non-blocking mode for the completion of a given call to complete. Being based on a client-server model, testing DIET-ethic in this context seemed a natural choice.

Let us present how GridRPC is implemented in DIET [7]. The DIET component architecture is hierarchically structured for improved scalability as illustrated in Figure 3. The DIET toolkit is implemented in CORBA [13] and thus benefits from the many standardized, stable services provided by freely-available and high performance CORBA implementations.

The DIET framework has several elements. A **Client** is an application that uses the DIET infrastructure to solve problems using a GridRPC approach. A **SeD** (**Server Daemon**) acts as the service provider, exporting functionality through a standardized computational service interface; a single SeD can offer any number of computational services.

The third element of the DIET architecture, **agents**, facilitate the service location and invocation interactions of clients and SeDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a **Master Agent (MA)** and several **Local Agents (LA)**.

A typical DIET interaction goes as follows: (1) First, the client enters the system and sends a discovery message with some information regarding the request to be computed to the master agent. (2) Next, the message is propagated from the master agent to the SeD through local agents. The message eventually reaches all SeDs, (3) which reply with an **estimation vector**: a set of values describing how fit they are for solving this particular request. Depending on the implementation of a service, an estimation vector might contain information such as CPU power, amount of RAM, estimated completion time of the request, number of queued requests, etc. (4) At each level of the agent hierarchy the estimation vectors are aggregated, so that the master agent only has to deal with a small amount of them. (5) Finally, one or more estimation vectors are returned to the client, which chooses a suitable SeD. (6) The request with all the data necessary to solve the problem is sent directly from the client to the SeD.

Let us highlight where scheduling decisions are taken. First, SeDs decide what values to put in an estimation vector. This way, a SeD can make itself more or less preferred for solving a problem. Second, agents throughout the hierarchy aggregate and filter SeDs based on these estimation vectors. For example, only the top 10 least loaded SeDs are returned to the parent agent, thus leaving out overloaded SeDs. Third, clients receive estimation vectors from several SeDs, out of which they may choose a SeD.

To sum up, DIET already implements many prerequisites, such as service calls, scalable scheduling and data management. This allowed us to implement DIET-ethic with minimal effort.

4.3 Optional Computation Requests in Diet

Besides the addition of a request dispatcher and a queuing module, the following changes had to be made to implement DIET-ethic on top of DIET.

First, three extensions to the GridRPC API have been needed. (1) The `grpc_discover` function allows clients to discover servers in the system. The function accepts two parameters: the name of the service and the maximum number of servers to return. (2) The `grpc_function_handle_set_optional` function allows clients to declare a request as optional. (3) The `grpc_wait_any_until` function has been implemented, which is an extension to `grpc_wait_any` allowing a client to specify a timeout. The function block until either a GridRPC request completes or the timeout expires, similarly to the POSIX™ `select` or `poll` system calls.

Second, the client-server communication protocol had to been extended, so that each request is tagged with the user it belongs to. This is necessary information allowing servers to ensure fairness among different users. The current implementation assumes that users are honest and do not attempt to falsify their identity. A future implementation will include authentication, thus making sure that requests can be securely associated with a user.

Thus, we obtained a production-ready implementation which we shall use to evaluate the Diet-ethic architecture in the following section.

5 Evaluation

This section evaluates the proposed architecture. First, we show the gains that can be made with Diet-ethic by comparing it to a standard system which has not been designed to support optional computations. Second, we show that the architecture is scalable. We would like to highlight that all experiments have been done on a **real platform**. Finally, we discuss the benefits that can be observed by an end-user.

5.1 Gains of Supporting Optional Computations

Let us consider increasingly complex scenarios and make a comparative analysis between Diet-ethic and a system without optional computation support. To this end, we used the Diet middleware as it was before our contribution: the SeDs serve incoming requests using the FCFS policy, without distinguishing mandatory from optional requests. On the client-side, we implemented the following behaviour. When a client i enters the system it has to blindly choose $n^{(i)}_{submit}$, a number between $n^{(i)}_{min}$ and $n^{(i)}_{max}$, representing the number of requests to submit. First, it submits $n^{(i)}_{min}$ mandatory and $n^{(i)}_{submit} - n^{(i)}_{min}$ optional requests (in this order). Then, it waits for the mandatory requests to finish. If the deadline has not expired, it sleeps until the deadline is reached. Finally, it gathers the results of all completed requests and cancels the remaining optional requests submitted to the system. To simplify the analysis of the results, all clients "guess" the same value n_{submit}. For briefness, this system shall be call the **legacy system**.

Before detailing the scenarios, let us present the common methodology. The platform consists of 1 MA and $n_R = 10$ SeDs. The SeDs only implement a sleep service, i.e., the service itself consumes no CPU nor network bandwidth. The platform is used by $n_U = 10$ identical clients with their parameters chosen as follows: in order to make experiments as useful as possible, but at the same time reduce the time it takes to complete them, we have chosen to "compress" the time: 8 hours are normalized to 100 s. Therefore, we set the execution time $T^{(i)} = 1$ s and the deadlines $d^{(i)} = 100$ s. These values are large enough compared to the time it takes to take scheduling decisions, yet small enough so that the time of experiments be reasonable. Next, we set the number of mandatory requests $n^{(i)}_{min} = 3$ and the number of total requests $n^{(i)}_{max} = 1000$. These parameters have been chosen so that the following conditions be met:

- there is a solution which makes all users happy;
- each user can generate enough optional computations to fill all resources.

The above conditions are the ones in which our system is the most interesting to be studied. Otherwise, if the number of mandatory computations is too high, the platform has no choice but to schedule them in a FCFS fashion, being forced to make some users unhappy. Also, if the number of optional requests is too low, its fairness properties cannot be highlighted.

The metrics we are interested in are those presented in Section 3.3. All measurements have been done at least 10 times and, since we found deviations to be small, we chose to make graphs more readable by only plotting the median.

The systems are compared in 5 different, increasingly complex scenarios, which are described in detail in the next sections.

5.1.1 Night-time Simultaneous Submissions

Let us start with a simple scenario. Users want to do computations during the night, so that their re-

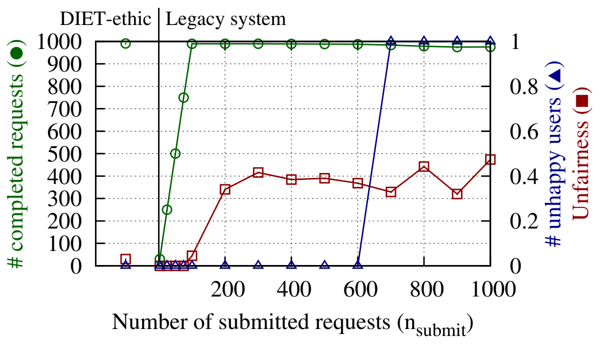

Figure 4: Results: night-time simultaneous submissions scenario

sults are ready in the morning and can be analysed during the workday. Effectively, users enter the computation platform in the evening, just before leaving work and have a tentative deadline for the next morning, when they arrive at work. In our experiments, we can model them by setting the same arrival-time $t_0^{(i)} = 0$ and deadline $d^{(i)} = 100$ for all users.

Figure 4 shows that DIET-ethic managed to find a solution with no unhappy users, with good (almost ideal) fairness, while maximizing the number of completed requests. Regarding the legacy system, one observes that, if n_{submit} is small, the resources are not filled with computation requests, thus, the number of completed requests is suboptimal. However, if n_{submit} is high enough, on average, the legacy system behaves fairly well. This is due to an experiment artifact that, since all users enter the system at precisely the same moment of time, their requests favorably interleave, therefore, the FCFS policy is mostly finding the optimal solution: all mandatory requests are started first, followed by the optional requests.

5.1.2 Night-time Consecutive Submissions

However, in production systems, users never enter the platform at exactly the same time. In fact, the time the users enter the system might be quite different: some people leave work earlier, others later. Their deadlines are about the same, since those who leave work earlier, often come earlier the next day. To model this scenario, we insert a very small inter-arrival gap $t_0^{(i)} = i \cdot 0.1\,\text{s}$ and keep $d^{(i)} = 100\,\text{s}$.

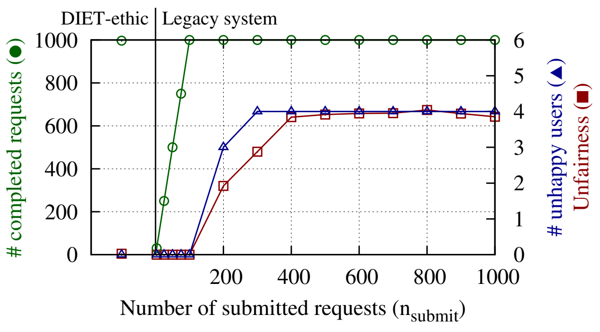

Figure 5: Results: night-time consecutive submissions scenario

Figure 5 shows that, unless all users guess the ideal solution (that of each user submitting exactly $n_{ideal} = 100$ requests), the legacy system either does not manage to optimize the number of completed requests (if $n_{submit} < n_{ideal}$) or makes users unhappy (if $n_{submit} > n_{ideal}$). The latter happens because the FCFS policy fills resources with optional requests of users who arrived early in the system. Therefore, the mandatory requests of users who arrive later start later and can be delayed past the tentative deadline. A similar observation applies to fairness: the FCFS policy favors users who enter the system early, instead of trying to balance requests equally among them.

In contrast, since DIET-ethic distinguishes mandatory and optional requests, it makes sure that mandatory requests have priority over optional ones. Also, instead of favoring users who arrive early, resources are allocated equally among the optional requests of the users. In the end, DIET-ethic improves fairness up to 150 times and behaves as if all users chose the ideal number of requests to submit, but without requiring the users to guess this value.

5.1.3 Day-time Submissions with Regular Arrivals

Let us pass on to a different scenario: day-time submissions. During the day, the users to not enter the system during a short time interval, but are separated by significant inter-arrival times. To start, we study a simpler scenario, in which the inter-arrival time between consecutive clients is constant.

To model this, we took the previous scenario and

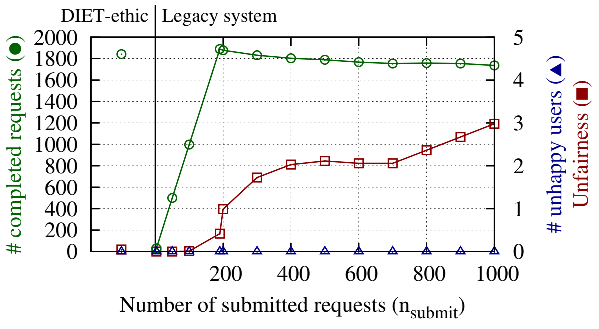

Figure 6: Results: day-time scenario with regular arrivals

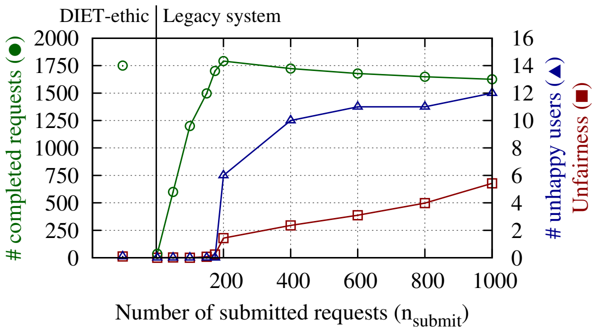

Figure 7: Results: day-time scenario with irregular arrivals

set $t_0^{(i)} = i \cdot 10\,\mathrm{s}$. Except arrival times, all other experimental parameters are kept the same. When contrasting the two scenarios, the main difference is that, in the previous one all clients have entered the system before the mandatory requests of the first client are completed. In contrast, in the current scenario, mandatory requests of a client are already completed by the time the next client arrives in the system. One could say that the previous scenario resembles an off-line scheduling problem (i.e., all requests are known in advance, decisions can be taken in advance), whereas the current scenario resembles an on-line scheduling problem (i.e., the system needs to adapt to arriving requests).

Figure 6 shows the results for this scenario. One can observe that the legacy system behaves best for $n_{submit} = 190$. No users are unhappy, unfairness is low and the number of completed requests is the highest, even when compared to DIET-ethic. The latter happens because, when a new client enters the system, DIET-ethic immediately starts its mandatory requests, killing optional request if necessary. The so-far done computations are not completed, thus reducing the number of completed requests. Nevertheless, one observes that DIET-ethic's solution has a lower (near-ideal) unfairness and the number of completed requests stays competitive to the legacy system (1842 vs. 1890: $\approx$ -2.5%).

However, on a real platform guessing the best number of requests $n_{submit}^{(i)}$ each client i should submit is difficult, as it depends on a number of factors, such as the number of resources, arrivals and requirements of other users. Some of this information is unknown at the time a client enters the system.

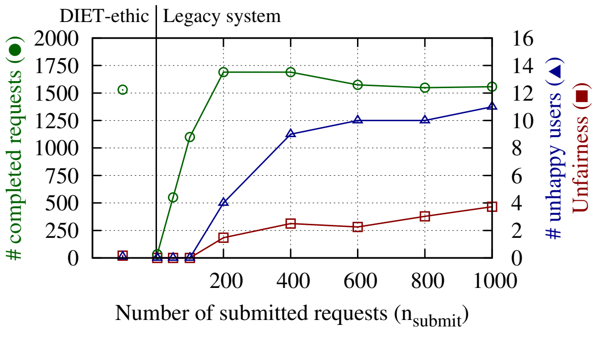

Figure 8: Results: irregular arrivals and random execution times

When looking at the results for $n_{submit} \neq 190$, one observes that the legacy system is outperformed by DIET-ethic. As in previous scenarios, if fewer requests are submitted, then the number of completed requests is suboptimal. Deviating in the other direction, if too many requests are submitted then unfairness increases.

Interestingly, as the number of submitted requests increases, the number of completed requests slightly decreases. This happens because, increasing n_{submit} also increases the probability that clients launched earlier still have executing requests in the system by the time their deadline is reached. These requests are killed, thus are not completed.

In contrast, DIET-ethic auto-tunes itself and finds a good solution, without requiring the user to guess a good number of requests to submit.

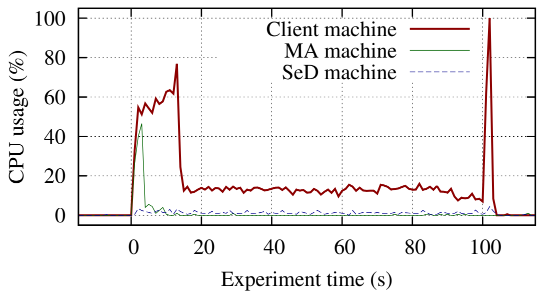

Figure 9: System overhead: CPU usage for sleep requests

Table 1: Grid'5000 deployment for scalability experiment

Cluster	# nodes	Configuration
capricorne	2	2×AMD Opteron 246 @ 2.0 GHz
sagittaire	69	2×AMD Opteron 250 @ 2.4 GHz

Table 2: Results of scalability experiment

Unhappy users	0	ideally 0
Unfairness	0.70	ideally 0
Number of completed requests	13605	out of 13800

5.1.4 Other Day-time Scenarios

In order to make sure that the proposed system is well-behaved in more realistic cases, let us present two more scenarios.

First, we drop the assumption that the inter-arrival time between consecutive clients is constant. Instead, the arrivals are assumed to obey the well-known and widely recognized diurnal cycle [14]. To model this, we took the polynomial P proposed in [15], scaled its input to the $[0, 100]$ seconds interval and used the output value as follows: at each second t, the probability of launching a new client is $P(t)/10$. The division by 10 was made so as to keep the number of clients in the system of the same order-of-magnitude as in the other scenarios.

Second, we give each client a different execution time, by choosing $T^{(i)}$ uniform randomly in $[0.75, 2]$. Requests generated by the same client are still homogeneous. The arrivals are considered to obey the diurnal cycles as in the above paragraph.

For both of these scenarios (figures 7 and 8), the same observations can be made as in previous sections. There is a value of n_{submit} in which the legacy system behaves well, however, this value is difficult to compute a priori in a real system. In contrast, DIET-ethic auto-tunes itself and manages to minimize the number of unhappy users, minimize unfairness and maximize the number of completed requests.

5.2 Scalability

In order to assess the scalability of our solution and measure the overhead, we have designed the following experiment. We reserved the **whole** Lyon site on the Grid'5000 experimental platform [16] (Table 1). The set of nodes has been divided into three: 1 client node, 1 MA node and 69 SeD nodes.

Experiments has been done as follows: first, 1 MA has been deployed on the MA node. Second, SeDs have been deployed on each core of the bi-processor SeD nodes, totalling $n_R = 2 \times 69 = 138$ SeDs. Finally, on the client node, $n_U = 100$ client have been launched simultaneously with the parameters: $n_{min}^{(i)} = 3$, $n_{max}^{(i)} = 10000$, $d^{(i)} = 100$, $T^{(i)} = 1$. As a reference, traces from the Grid Workload Archive [17] contains less that 100 users per day.

Besides the metrics presented in Section 3.3, we measure the CPU utilization on the client node, the MA node and one of the SeD nodes. On the client node, we have been careful to filter out CPU usage due to process creation and destruction. The SeDs implement a simple "sleep" service, which does not do any computations. Therefore, the measured CPU usage represents the **overhead** our system incurs for managing computational requests.

The metrics can be found in Table 2. They show that there are no unhappy users and that unfairness is low. Also, the number of completed requests is 13605, which, compared to the maximum of 13800 requests that could have been completed by 138 SeDs in the 100 s deadline, represents 98.6%. Hence, we conclude that the system managed to optimize the targeted metrics, even under stress conditions.

Let us now take a deeper look at the behaviour of the system and study the CPU usage overhead during the experiments (Figure 9). First, let us observe that the measured CPU usage before launching the clients and after the clients finished is below 1%, thus indicating a low measurement noise.

Next, one can clearly distinguish the three phases of the clients (Section 4.1.2): setup, monitoring and cleanup.

The *setup* phase takes about 13 s from $t = 0$ to $t = 13$. During this phase the CPU usage on the client node is somewhat high, due to all clients simultaneously discovering SeDs, then submitting requests to all of them. Discovery is handled by the MA, on which one observes a peak in CPU usage. On the SeD node, the CPU usage is negligible, being less than 3%.

During the *monitoring* phase, which lasts from $t = 13$ to $t = 100$, one observes that the CPU usage stays relatively low on all nodes. The most stressed is the client node, as the client applications have to submit 138 requests/s to keep SeDs busy. The MA is idling, since it is not participating in this phase, while the CPU usage on the SeD node is negligible.

Finally, during the *cleanup* phase, the CPU usage on the client node tops at 100%, as cancellation requests are sent by all clients, simultaneously to all SeDs. Again, the MA is idling, while the CPU usage on the SeD node is still fairly low ($< 5\%$).

Note that the scalability experiment is extreme. The clients arrive simultaneously, which is unlikely in real systems and the execution time is short (1 s). For comparison, the average job inter-arrival time on the LCG grid is 5 s, whereas the average runtime is 2.5 h [17].

To sum up, the CPU usage on the SeD nodes is negligible when a sleep service is used. This means that our system involves **low overhead** and that SeDs can perform useful computations. The CPU usage on the client node is high during the setup and cleanup phase, nevertheless it managed to generate enough requests, so as to keep SeDs busy. The MA did not prove to be a bottleneck in these experiments.

5.3 End-User Perspective

Now that we evaluated DIET-ethic with a synthetic "sleep" service, we return to the motivating application (Section 2) and discuss the advantages that an average end-user observes. We have implemented a working prototype in order to test an initial version of DIET-ethic against GRID-TLSE.

Let us briefly describe what changes were necessary. GRID-TLSE is composed of an upper layer ALTO, responsible for interpreting high-level user requests and aggregating results, and a lower layer BASSO, responsible for communicating with a GridRPC middleware and performing computations. ALTO had to be extended so as to accept a variable number of results for every request sent to BASSO. Previously, ALTO was coded so as to accept a pre-computed number of results for each request, which is no longer true, since the number of completed requests (and, as a consequence, the number results returned to ALTO) depends on the status of the platform. As to BASSO, the behavior described in Section 4.1.2 has been implemented.

Previously, when a user wanted to solve a linear system, she had to choose the number of thresholds to test. This was done using a combination of trial-and-error and guesswork. The user had to know approximately the number of available resources and the time a solve takes. If the user overestimated the number of thresholds, she would manually cancel the computations. Conversely, the user underestimated, she would have to manually relaunch the computations with a new set of thresholds.

In contrast, thanks to our contribution, a working prototype showed that the user only has to specify the minimum and the maximum number of thresholds to test, and a deadline. The system automatically tests as many thresholds as to respect the constraints imposed by the users.

6 Related Work

This section reviews related work from areas of real-time systems and malleable applications.

6.1 Real-time Systems

A system is considered **real-time** if it needs to guarantee a certain response time, i.e., there are tasks that need to be completed before a given deadline. Based on their constraints, real-time systems can either be **hard**, i.e., missing a deadline leads to system failure, or **soft**, some deadlines can be missed. Furthermore, the generation of the tasks can be either **sporadic**, i.e., the arrival time of tasks is not known in advance, or **periodic**.

For each case, various algorithms have been proposed. For example, Earliest Deadline First (EDF) is an algorithm for hard real-time systems with sporadic tasks. The next task to be executed is the one that is closest to its deadline. It has been shown [18] that under certain conditions, EDF is optimal: if there exists such a solution, EDF always find a schedule so that all of the tasks meet their deadlines. However, EDF is considered to be expensive to compute, so more efficient ways of computing a schedule have been studied. If tasks are periodic, one can pre-compute an efficient schedule by simulating the behavior of EDF for a certain period of time. Moreover, having information about all future tasks allows to find a schedule that is not only valid, but also fair [19].

This article shares many concepts with the cited works, such as guaranteeing deadlines and improving fairness. However, one of our base assumptions is that there are not enough resources to execute all tasks in the given deadlines. The question is then which tasks to sacrifice, in order to keep users satisfied.

Let us now review the field of soft real-time scheduling. The (m, k)-firm deadlines problem [20] consists in finding a schedule in a real-time system with periodic tasks, so that at least m out of k consecutive tasks meet their deadline. Various algorithms have been proposed that basically divide the tasks into mandatory and optional [21].

Our problem shares some vocabulary (mandatory and optional), but differs in several ways. First, arriving tasks are not periodic. In fact, both their arrival-time and their execution-time are unknown to the platform. Second, it is the user who decides which tasks are to be considered mandatory and which are to be considered optional. Third, the deadline is considered for a group of tasks (i.e., those belonging to the same user) and not for every task individually. Therefore, the solutions brought by cited works are not directly applicable to our problem statement.

6.2 Malleable Applications

An application is said to be **malleable** if it can change its resource usage as requested by the scheduler [22]. This can be done in order to improve resource utilization or decrease response times [23]. How to write malleable applications [24, 25] and how to add scheduler support for them [26, 27, 28] has been extensively studied.

Our solution can be considered yet another scheduling problem that is addressed using malleability. However, all of the cited works aim at using malleability to improve some kind of metric, but, in the end, the computations that are executed are always the same, no matter how resources are allocated to the applications. In contrast, to our knowledge, we are the first to propose for a malleable application to adapt to the state of the resources by also changing the computations it does. If many resources are available, the application produces more accurate results, whereas if few resources are available, the application guarantees at least a minimum accuracy.

Parameter-sweep applications are one of the most common classes of malleable applications [10]. They consist of many sequential tasks, which are relatively short when compared to the execution-time of the whole application. On one hand, this is an advantage, since the scheduling granularity is relatively small, allowing much freedom. For example, a task can be killed and restarted later, possibly on another resource, without losing significant computations. On the other hand, due to the small granularity, one has to take extra precautions not to overload the system.

To solve the above issue, the DIRAC workload management system [29] proposes a solution based on the pilot job abstraction. **Pilot jobs** are container jobs that, instead of executing an actual task, launch a daemon. Once the job starts and the daemon is running, the latter connects to a centralized agent and requests tasks to execute. In DIRAC, the tasks are put in a central queue and the scheduling event is initiated by resource availability. One can say that computation requests are *pulled* by the resources from the centralized agent. This is in contrast to traditional usage of HPC platform (e.g., imposed by batch schedulers), in which jobs are *pushed* to resources. DIRAC's approach leads to a natural load balancing where each resource gets the workload matching its actual capacity. In a push approach this is usually achieved by ranking resources based on their capacity, which requires complex calculations and a reliable information system.

Our contribution borrows some concepts both from push and pull scheduling. Mandatory requests are pushed from clients to resources, whereas optional requests are pulled by resources from clients. The former allows clients to dispatch mandatory requests to the most fitting resources, whereas the latter allows to reduce the strain on the platform and produce a natural load balancing as done by DIRAC.

Two aspects differentiate DIET-ethic from DIRAC. First, a limited number of computation requests is stored on the platform, the rest being generated on-the-fly by the clients. In contrast, DIRAC requires all computational requests to be submitted to a centralized agent. Our design choice is necessary to ensure the scalability of the system, since it is possible that only a small percentage of optional requests are actually executed, thus, storing them all centrally would overload the system. Second, DIRAC takes most scheduling decisions in a centralized agent. In contrast, in DIET-ethic, each client and each server contributes to taking scheduling decisions. This can be considered an advantage, since the two entities can evolve their scheduling algorithms separately. For example, a user wanting to implement a different scheduling algorithm can change the client application, without having to wait for a platform upgrade.

7 Conclusion

This paper presented DIET-ethic a platform architecture to efficiently support optional computations. It can easily be implemented in any GridRPC middleware, such as GridSolve [30] or Ninf [31]. Evaluation has been done using an implementation on top of the production-level DIET middleware. Real-life experiments showed that several metrics can be improved, for example user unhappiness can be reduced to 0 and unfairness can be decreased up to 150 times. Finally, the architecture was shown to be scalable.

As future work we propose improving the scheduling of mandatory requests by reallocating them to better suited servers [32], instead of mapping them at submittal as currently done. Scheduling of optional computations could be improved by using preemption [33]. Different window sizes would have to be tested to find the best scheduling.

Acknowledgments

This work is supported by the French ANR COOP project, n°ANR-09-COSI-001 (`http://coop.gforge.inria.fr`). Experiments were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (`https://www.grid5000.fr`). Special thanks to Gilles Fedak for his insightful comments.

References

[1] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling - a status report," in *JSSPP*, ser. Lecture Notes in Computer Science, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., vol. 3277. Springer, 2004, pp. 1–16.

[2] E. Caron, V. Garonne, and A. Tsaregorodtsev, "Definition, modelling and simulation of a grid computing scheduling system for high throughput computing," *Future Generation Computer Systems*, vol. 23, no. Issue 8, pp. 968–976, November 2007, iSSN:0167-739X.

[3] J. Helton, J. Johnson, C. Sallaberry, and C. Storlie, "Survey of sampling-based methods for uncertainty and sensitivity analysis," *Reliability Engineering & System Safety*, vol. 91, no. 10-11, pp. 1175–1209, Oct. 2006.

[4] "Amazon ec2 spot instances." [Online]. Available: http://aws.amazon.com/ec2/spot-instances/

[5] "High performance computing (HPC) on Amazon web services." [Online]. Available: http://aws.amazon.com/hpc-applications/

[6] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of GridRPC: A remote procedure call API for grid computing," in *Grid Computing - GRID 2002, Third International Workshop*, ser. LNCS, M. Parashar, Ed., vol. 2536. Baltimore, MD, USA,: Springer, Nov. 2002, pp. 274–278.

[7] E. Caron and F. Desprez, "DIET: A scalable toolbox to build network enabled servers on the grid," *International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 335–352, 2006.

[8] P. R. Amestoy, I. S. Duff, L. Giraud, J.-Y. L'Excellent, and C. Puglisi, "Grid-TLSE: A web site for experimenting with sparse direct solvers on a computational grid," in *SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

[9] N. Bard, R. Bolze, E. Caron, F. Desprez, M. Heymann, A. Friedrich, L. Moulinier, N. H. Nguyen,

O. Poch, and T. Toursel, "Décrypthon grid - grid resources dedicated to neuromuscular disorders," *Studies in Health Technology and Informatics*, vol. 159, pp. 124–133, 2010.

[10] O. O. Sonmez, B. Grundeken, H. H. Mohamed, A. Iosup, and D. H. J. Epema, "Scheduling strategies for cycle scavenging in multicluster grid systems," in *CCGRID*, 2009, pp. 12–19.

[11] Y. Yuan, G. Yang, Y. Wu, and W. Zheng, "PV-EASY: a strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 240–251.

[12] "Open grid forum." [Online]. Available: http://www.ogf.org/

[13] Object Management Group, "Common object request broker architecture (CORBA/IIOP) – version 3.2," Specification, Nov. 2011. [Online]. Available: http://www.omg.org/spec/CORBA/3.2/

[14] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*, version 0.34 ed., 2011. [Online]. Available: http://www.cs.huji.ac.il/~feit/wlmod/

[15] M. Calzarossa and G. Serazzi, "A characterization of the variation in time of workload arrival patterns," *IEEE Trans. Comput.*, vol. C-34, no. 2, pp. 156–162, Feb 1985.

[16] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed." *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, Nov. 2006.

[17] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema, "The grid workloads archive," *Future Generation Comp. Syst.*, vol. 24, no. 7, pp. 672–686, 2008.

[18] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," in *IFIP Congress*, 1974, pp. 807–813.

[19] S. K. Baruah, "Fairness in periodic real-time scheduling," in *IEEE Real-Time Systems Symposium*, 1995, pp. 200–209.

[20] P. Ramanathan and M. Hamdaoui, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Trans. Comput.*, vol. 44, no. 12, pp. 1443–1451, Dec. 1995.

[21] G. Quan and X. Hu, "Enhanced fixed-priority scheduling with (m,k)-firm guarantee," in *Proceedings of the 21st IEEE conference on Real-time systems symposium*, ser. RTSS'10. Washington, DC, USA: IEEE Computer Society, 2000, pp. 79–88.

[22] D. G. Feitelson and L. Rudolph, "Towards convergence in job schedulers for parallel supercomputers," in *JSSPP*, 1996.

[23] J. Hungershofer, "On the combined scheduling of malleable and rigid jobs," in *SBAC-PAD*, 2004, pp. 206–213.

[24] K. El Maghraoui, T. J. Desell *et al.*, "Dynamic malleability in iterative MPI applications," in *CCGRID*, 2007, pp. 591–598.

[25] J. Buisson, F. André *et al.*, "A framework for dynamic adaptation of parallel components," in *PARCO*, 2005, pp. 65–72.

[26] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema, "Scheduling malleable applications in multicluster systems," CoreGRID, Tech. Rep. TR-0092, 2007.

[27] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard *et al.*, "Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI," in *ICDCN*, 2010.

[28] R. Sudarsan and C. J. Ribbens, "ReSHAPE: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment," *CoRR*, vol. abs/cs/0703137, 2007.

[29] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev, and the LHCb DIRAC Team, "DIRAC pilot framework and the DIRAC workload management system," *Journal of Physics: Conference Series*, vol. 219, no. 6, 2010.

[30] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra, "Recent developments in GridSolve," *IJHPCA*, vol. 20, no. 1, pp. 131–141, 2006.

[31] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-g: A reference implementation of rpc-based programming middleware for grid computing," *J. Grid Comput.*, vol. 1, no. 1, pp. 41–51, 2003.

[32] Y. Caniou, G. Charrier, and F. Desprez, "Analysis of Tasks Reallocation in a Dedicated Grid Environment," in *IEEE International Conference on Cluster Computing 2010 (Cluster 2010)*, Heraklion, Crete, Greece, September 20-24 2010, pp. 284–291.

[33] J. Li, M. Qiu, Z. Ming, G. Quan, X. Qin, and Z. Gu, "Online optimization for scheduling preemptable tasks on IaaS cloud systems," *Journal of*

Parallel and Distributed Computing, vol. 72, no. 5, pp. 666–677, May 2012.

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399