Available online at www.sciencedirect.com

ScienceDirect

Future Generation Computer Systems 24 (2008) 331–341

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/fgcs

Allocation strategies for utilization of space-shared resources in Bag of Tasks grids[☆]

César A.F. De Rose[a], Tiago Ferreto[a], Rodrigo N. Calheiros[a], Walfredo Cirne[b,*], Lauro B. Costa[b], Daniel Fireman[b]

[a] *Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil*
[b] *Universidade Federal de Campina Grande, Campina Grande, Brazil*

Received 28 January 2007; received in revised form 23 May 2007; accepted 29 May 2007
Available online 15 June 2007

Abstract

As the adoption of grid computing in organizations expands, the need for wise utilization of different types of resource also increases. A volatile resource, such as a desktop computer, is a common type of resource found in grids. However, using efficiently other types of resource, such as space-shared resources, represented by parallel supercomputers and clusters of workstations, is extremely important, since they can provide a great amount of computation power. Using space-shared resources in grids is not straightforward since they require jobs a priori to specify some parameters, such as allocation time and amount of processors. Current solutions (e.g. Grid Resource and Allocation Management (GRAM)) are based on the explicit definition of these parameters by the user. On the other hand, good progress has been made in supporting Bag-of-Tasks (BoT) applications on grids. This is a restricted model of parallelism on which tasks do not communicate among themselves, making recovering from failures a simple matter of reexecuting tasks. As such, there is no need to specify a maximum number of resources, or a period of time that resources must be executing the application, such as required by space-shared resources. Besides, this state of affairs makes leverage from space-shared resources hard for BoT applications running on grid. This paper presents an Explicit Allocation Strategy, in which an adaptor automatically fits grid requests to the resource in order to decrease the turn-around time of the application. We compare it with another strategy described in our previous work, called Transparent Allocation Strategy, in which idle nodes of the space-shared resource are donated to the grid. As we shall see, both strategies provide good results. Moreover, they are complementary in the sense that they fulfill different usage roles. The Transparent Allocation Strategy enables a resource owner to raise its utilization by offering cycles that would otherwise go wasted, while protecting the local workload from increased contention. The Explicit Allocation Strategy, conversely, allows a user to benefit from the accesses she has to space-shared resources in the grid, enabling her natively to submit tasks without having to craft (time, processors) requests.

Keywords: Computational grids; Resource management; Space-shared resources; Bag-of-Tasks

1. Introduction

Grid computing has enticed many with the promise to allocate unprecedented amounts of resources to a parallel application, and to make it with lower cost than traditional alternatives (based on parallel supercomputers) [1–4]. However, not all parallel applications are equally suited for execution in grids. Bag-of-Tasks (BoT) is an application model that is especially suitable for execution in grids since it is composed of several uniprocessor tasks that demand no communication during its execution, tolerating network delays and faults. These characteristics facilitate the utilization of volatile resources in the grid, i.e. computational resources that join and leave the grid with no previous notice, have unknown and varying power and may return incorrect results. In order to achieve good performance with this type of resource, an eager scheduler [5–7] can be used. It uses task replication to tolerate computational power variability without relying on resource performance forecasts.

[☆] This work was developed in collaboration with HP Brazil R&D.

[*] Corresponding address: Departamento de Sistemas e Computacao, Universidade Federal de Campina Grande, Campina Grande, Brazil. Tel.: +55 83 3310 1365.

E-mail addresses: cesar.derose@pucrs.br (C.A.F. De Rose), walfredo@dsc.ufcg.edu.br (W. Cirne).

0167-739X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2007.05.005

However, space-shared resources do not match well with the definition of volatile resources, thus making it very hard for a scheduler that expects volatile resources to use space-shared resources. This is unfortunate because space-shared resources (such as parallel supercomputers and clusters of workstations) are among the most powerful resources available in a grid, and could greatly expedite the execution of BoT applications.

Space-shared resources are used through a formal job submission to the resource scheduler specifying the number of processors needed and the amount of time these processors should be allocated to the incoming job. This job submission interface becomes a problem for grid users to execute their loosely coupled applications using space-shared resources. Besides, eager schedulers are not prepared to craft such a request, since they assume that all they have to do is to send a task that may eventually be executed by the resource. The current way to make space-shared resources available to eager schedulers consists of delegating the formal job submission to the user. However, this approach presents important performance and scale limitations considering that a grid may contain many different space-shared resources.

We present in this paper the Explicit Allocation Strategy, which aims to use space-shared resources efficiently in a grid. It uses heuristics automatically to craft formal space-shared requests from grid-brokers' requests in order to provide resources to grid users. This strategy is related to our previous work on automated strategies for using space-shared resources in grids called the Transparent Allocation Strategy [8]. The goal of this strategy was to use opportunistic computing techniques providing idle resources from space-shared resources to grid users.

It is important to realize that these strategies are complementary, in the sense that they play very different roles within a grid. The Transparent Allocation Strategy enables a resource owner to raise its utilization by offering cycles that would otherwise go wasted, while protecting the local workload from increased contention. Clearly, a job running opportunistically via the Transparent Allocation Strategy has a lower quality of service than a space-shared job (i.e. one which specifies a formal (time, processors) request). Moreover, not all jobs can benefit with the same ease from opportunistic resources. However, such resources can be very useful for BoT applications.

On the other hand, a user of a BoT application may also have access to a number of space-shared resources within the grid, and she might want her job to run faster than "opportunistically", taking advantage of the better quality of service her formal requests enjoy. However, different space-shared resources have dissimilar characteristics, and she does not want manually to craft requests for each of these resources. She only wants to run her job at the highest speed possible, using whatever space-shared resources she can access, as well as whatever opportunistic resources become available. The Explicit Allocation Strategy allows her *transparently* to benefit from space-shared resources she can access, enabling her natively to submit tasks without having to craft resource-specific (time, processors) requests.

Both strategies were implemented in OurGrid [9] and validated using different simulation scenarios. Our analysis concluded that the two strategies are complementary, providing two distinct qualities of service, relying on space-shared resources utilization characteristics and policies.

The rest of this paper is organized as follows: Section 2 presents how a space-shared scheduler works and some issues related to its utilization in grids. Section 3 describes the Transparent Allocation Strategy, whereas Section 4 evaluates the Explicit Allocation Strategy. Section 5 presents a comparison of both strategies and an analysis of using both strategies together. Finally, Section 6 presents our conclusions and future work.

2. Space-shared resources

Space-shared resources, such as distributed-memory parallel supercomputers or clusters of workstations, are high-end machines designed to support the execution of parallel applications, promoting its performance. In this architecture, a parallel application receives a dedicated partition of resources for exclusive utilization. In order to obtain this partition, it is necessary to perform a formal request specifying p, the number of processors to be allocated to the application, and tr, the time requested for the execution of the application. This request is sent to a space-shared resource scheduler, which manages the space-shared resource and provides the access to a dedicated set of resources.

There are a handful of space-shared resource schedulers currently in production. These include Easy [10], PBS [11], Crono [12] and Maui [13] schedulers. In practice, however, the behavior of such schedulers varies from site to site. Even when the same scheduling software is used, each site configures its own policies, causing the behavior of their schedulers to differ. Therefore, we used a conservative backfilling approach as a good representative of today's schedulers and as a scheduler that is accepted as attaining good performance [14,15]. The main idea is that an arriving job is inserted into the first queue hole it fits. If the first job in the queue cannot be executed because there are not enough processors, the scheduler sweeps the queue looking for the first request that (i) can be executed with current available resources (free processors) and (ii) does not delay the start of any job in the queue. Such an approach guarantees predictability, giving an upper-bound to the job completion.

Regarding the utilization of space-shared resources in grids, commonly, request submissions to the grid for resources are not performed manually by users. Instead, discovering grid resources and submitting user's tasks in such resources is performed by grid schedulers [16] (typically, grid brokers), usually implementing an eager scheduling policy. Currently, examples of grid schedulers include Condor-G [17], Nimrod/G [18], GridWay [19] and MyGrid [20]. These systems perform task scheduling using some heuristic in order to optimize the overall application execution. Due to the variety of space-shared resources in a grid, and the diversity of interfaces provided by each scheduler, it is necessary to use an adaptor in the grid, which provides a standard interface to the grid user, and converts the request, with the number of processors and time, to the format specified by the chosen scheduler. Grid Resource and Allocation Management (GRAM) [21] is currently the best known implementation of such an approach, owing to the widespread utilization of the Globus Toolkit [22] to build grids.

However, the problem is that in the grid, users of BoT applications do not want to (in some cases, cannot) determine the number of processors and time to be requested. In this case, techniques such as performance models [23] and prediction models [24,25] can be used. However, while the latter does not make possible supplying a precise model for complex applications, the former requires an underlying software structure that is not always available to the prediction software.

Moreover, grid schedulers assume that all they need to do is to send a task to be executed in an available resource. Note also that the grid scheduler should not demand an explicit space-shared request from the user. Typically, a grid may contain many resources unknown to the user and user's runtime estimates are notoriously bad even when accessing homogeneous and known resources of a single supercomputer [26–29]. Thus, it is inappropriate to ask the user to estimate runtime (a key element of the request) in grid systems, as their resources are heterogeneous and unknown to the grid user.

To overcome such problems, in the next sections, we present two automated strategies to enable grid users to use space-shared resources in their applications. Section 3 presents the Transparent Allocation Strategy and Section 4 presents the Explicit Allocation Strategy.

3. Transparent allocation strategy

One of the challenges in grid computing is: "How to convince computing centers with a large amount of machines to donate their resources to a grid infrastructure?". The main excuses to not join a grid include: security problems, higher management costs, difficulty to manage and also account for resource utilization by grid users, among others. However, the main impact is observed in local users. Typically, users of large computing centers face the problem of sharing high performance resources with other local users. This usually results in delays, in terms of hours, or even days, to obtain the necessary resources. Enabling grid users to allocate these resources can increase even further this delay, resulting in greater discontentment of local users.

Despite the long queue resulting from local users requests, there are usually several fragments representing unused resources, that can be found in the queue. The amount of fragments is highly dependent on resources and requests characteristics. For example, the Horseshoe Bewolf cluster from the Danish Center for Scientific Computing presents an average idleness of 10%, which represents 80 CPU cores [30]. Instead of just losing this amount of computation power, it can be easily donated to the grid, without harming local users. Besides, current grid applications, more specifically BoT applications, do not need the same guarantees of dedicated resources and time, as parallel applications executing in space-shared resources.

The Transparent Allocation Strategy [8] is based on an idle cycles exploitation mechanism, i.e. when the space-shared resource is not fully allocated by local users, all remaining processors are transparently donated to the grid. The strategy prioritizes local users on behalf of grid users in relation to the guarantees for resource exclusive utilization, while simplifying resource access by grid users. This simplified access results from the utilization of space-shared resources as regular volatile ones.

Grid users access processors not in use in the space-shared resource through a grid scheduler, as made for regular volatile resources. The processors are not exclusively allocated to the grid user, i.e. there is no guarantee about the amount of time that the resource will be available to the grid. When a local user request needs to use processors being donated to the grid, the space-shared resource scheduler pre-empts and aborts all grid tasks being executed, in order to complete the local user's request, and removes the processors from the list of available resources in the grid. The grid scheduler handles this situation as a regular processor fault, and schedules the aborted tasks to other available resources, as usual.

One of the main goals of this strategy is the transparency of grid utilization to local users. Local users are not aware of resources utilization by grid users, since grid users do not perform a formal allocation of resources, and the processors donated to the grid are presented as idle in the space-shared resources scheduler queue. The strategy also does not change how local users use the resources. They still need to allocate resources performing an explicit request specifying the amount of processors p, and time tr needed. If the local user request needs processors in use by grid users, then the grid tasks are aborted in these processors, and they are removed from the list of available resources, as explained before.

The implementation of the Transparent Allocation Strategy requires a loosely coupled integration between space-shared resource scheduler and grid scheduler. In order to facilitate this integration, both components need to provide open interfaces. The main issues regarding the grid scheduler include: advertising available processors, removing "faulty" processors and aborting grid tasks. In the space-shared resource scheduler side, the main issue is to simplify the access and execution of tasks in the available processors.

One implementation of the Transparent Allocation Strategy is the CronoGuMP [8]. It interacts with the Crono [12] cluster resource manager in order to provide resources to the OurGrid [9] Community, which consists of several universities and research centers from Brazil [9,31]. Fig. 1 presents CronoGuMP. It is connected to Crono in order to identify when there are idle nodes that can be donated to the grid, and when local users want to use local resources, in order to take the resources off the grid. Resources can be in two states: being used by local users (marked with L in the figure) or by grid users (marked with G in the figure). When a resource

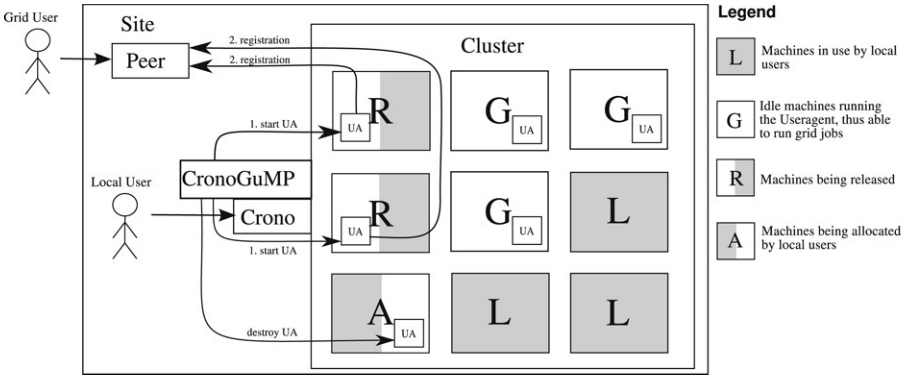


Fig. 1. Architecture of CronoGuMP.

is released (marked with R in the figure), CronoGuMP starts the OurGrid's agent module, called UserAgent (UA), on each released resource. Once started, the UserAgent contacts the OurGrid Peer to inform their availability, and the Peer makes these resources available to grid users.

When a local user requests resources (marked with A in the figure), CronoGuMP deactivates the UserAgent on each resource being allocated. Once the UserAgent is disabled, the resources are ready to be used. It is not necessary to inform the Peer about the change, as the Peer itself keeps track of grid resources availability. Thus, it will detect the resource loss and will remove it from the resource pool.

Results presented in [8] show that the Transparent Allocation Strategy is particularly useful for applications comprising a large number of short duration tasks to be executed in space-shared resources with medium or low load. Applications composed of medium duration tasks in these resources should experiment with an acceptable turn-around time. For those who need to execute large tasks or need to execute their applications in space-shared resources with a heavy load, another strategy must be applied.

4. Explicit allocation strategy

In this section we present the Explicit Allocation Strategy which implements a different approach from the Transparent Allocation Strategy. The previous strategy provides transparency and priority to local users. However, there are cases where a grid user has access (an account) to several space-shared resources and wants to use these resources at a higher local priority. This strategy's approach is based on adapting requests from Grid Scheduler to request resources to the space-shared resource scheduler.

Based on the factors presented in Section 2, the main problem in using space-shared resources with eager schedulers in a grid environment is such a detailed request. The choice of the parameters (*tr* and *p*) could render great impact in turn-around time [15]. In fact, several research works address the behavior of space-shared resources considering issues related to request area and backfilling [10,14,26,28,32,33] and the great impact that request area could cause in waiting time. On the other hand, a greater area allows more processing to be done. It is, thus, not clear whether one should issue a few large requests or many small ones.

4.1. Requested time issues

In order to specify a good value for *tr*, we should know the time needed to execute a grid task (a task sent by the grid broker or simply a *task*). Such a value should prevent us from: (i) crafting useless requests (in which the time is not enough to execute the task completely) or (ii) crafting big requests where significantly larger than needed (it may render long waiting times). This suggests that a way to specify this parameter is to ask the user how much time should be requested for a task (i.e., how much time is needed to execute a task). However, such an approach would be a quite difficult procedure since, in general, users do not have this kind of knowledge about the execution of their applications on every *space-shared* resource available on the grid.

Therefore, we propose that space-shared resource requests can be adapted by the grid middleware (request adaptor component) which estimates good values for time needed by one task. By adaptation we mean that parameters used to craft the requests can change dynamically such that new requests crafted could be better than old ones.

4.2. Number of processors issues

Coming up with the number of processors is easier than with the requested time, in the sense that a bad value for this parameter does not make impossible the task completion (that is, it does not render requests in which the time is not enough to execute the task completely). Nevertheless, it also impacts on how much faster a request could be processed. This happens because, as we have already mentioned, the execution start time of a space-shared job is directly related to its area (number of processors and amount of time requested); that is, a bigger area is harder to fit into the schedule, and hence tends to wait longer in the queue.

Note that a request for processors from a BoT grid broker can be broken into several independent space-shared requests, each one asking for fewer processors than the total number of needed processors (number of tasks) with no implication on the correct execution of the application. Therefore, each request

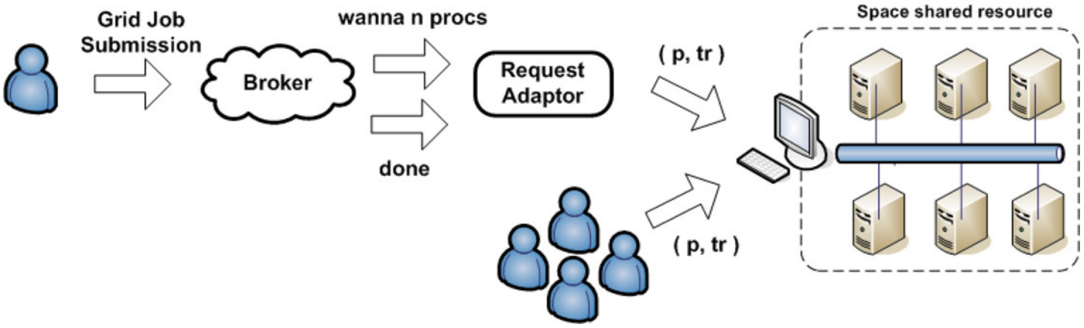


Fig. 2. Schedulers involved in running BoT applications on space-shared resources.

can fit into smaller free slots across the schedule and thus its execution can begin earlier. In the limit, one can issue many requests, each asking for a single processor.

However, all sites we know impose limits (via administrative policies) to the number of requests that a user can have in the system at any given moment. The idea seems to preclude "monopolization" of the system. This policy renders the specification of the number of nodes a harder task because we cannot simply submit lots of one-processor requests.

Therefore, when using a grid environment with space-shared resources and eager schedulers, the decision to craft requests must consider the relation between the number of processors needed, and the resource load and number of pending requests allowed in order to improve the turn-around time of the grid application.

4.3. *Automatically crafting requests*

Based on such a scenario and issues described in Sections 4.1 and 4.2, we propose an adaptive heuristic to craft the requests. The main goal of this heuristic is to convert a request for NoT processors from grid broker into several space-shared requests for (p, tr) providing a smaller turn-around time to the grid job (the collection of tasks). We intend to achieve such a goal trying to maximize task throughput (that is, the ratio between the number of finished tasks and the requested turn-around time).

Fig. 2 represents our model. There is a grid broker that receives the grid job from the user. The request adaptor receives the grid broker requirements (number of tasks to run) and tries to provide workers. This is done via a submission of space-shared requests crafted by heuristics. Each worker is a component (one per processor) that at a given moment runs only one task; that is, each processor can run several tasks during the requested time but only one at a given moment. The common use is a daemon that accepts grid broker invocations (e.g. MyGrid's UserAgent and Condor's Glide-In [17]).

Users submit their tasks to the grid broker which asks resource providers for processors. The request adaptor is one provider that elaborates space-shared requests (p, tr) and submits it to the resource scheduler queue. In the meantime, the space-shared resource has its local users that are also submitting space-shared requests.

In order to choose the parameters for requests, the request adaptor should obtain some information about the space-shared resource state (e.g. the requests in the queue), space-shared resource scheduler administrative policies (that impose some restrictions to requests) and the grid application. The information the request adaptor must know about the resource is as follows:

(1) the maximum allowed number of pending requests that grid broker can have on a space-shared resource scheduler (*maxPendingRequests*);
(2) the maximum allowed number of processors per request (*maxProc*);
(3) the maximum allowed amount of time requested per request (*maxTr*);
(4) the queue state.

The knowledge about grid applications is obtained by observing the execution of tasks; that is, the grid broker or request adaptor does not know anything about the grid job on its submission. As time goes by, it acquires knowledge about the application. The request adaptor saves (i) requested amounts of time, and (ii) whether the requested time of each task was enough for a task completion. Based on this information, the requested time is adapted. It can be enlarged (time was not enough) or shrunk (time was enough) for future requests.

4.3.1. *The heuristics*

The request adaptor uses heuristics to craft the requests. This section presents two heuristics to make requests: (i) *static*, a naive solution and (ii) *adaptive*, which makes requests based on previous task execution and on the state of space-shared resources. They are executed every time the grid broker asks for processors or when a space-shared request finishes.

The *static heuristic* asks for fixed requests of $nProcs = \frac{numberOfPendingTasks}{maxPendingRequests}$ processors and $tr = maxTr$ as requested time. Of course, *nProcs* should be an integer value (e.g. we cannot ask for 0.5 processor), so we use $\lceil nProcs \rceil$. If the number of tasks is too high (greater than the maximum possible number of processors requested), several new requests can be asked when old ones finish. The complete algorithm of the static heuristic is shown in Algorithm 1.

```
$allowedRequests \leftarrow maxPendingRequests() - actualPendingRequests()$;
for $i = 1$ to allowedRequests do
    $procs \leftarrow min(maxProc, \lceil remainingTasks/allowedRequests \rceil)$;
    $issueRequest(procs, maxTime)$;
end
```

Algorithm 1: Static heuristic.

The naive solution can provide good throughput; however, it uses the biggest possible area. As we have explained, big areas are harder to fit into a scheduling queue, thus rendering long turn-around times. In order to solve such problems, we propose the adaptive heuristic. Its adaptation occurs in both parameters, requested time and the number of processors; that is, the requests should be dynamically elaborated by learning from previous requests and the queue state of the resource providing better throughput.

To calculate throughput, we must estimate task runtime. The adaptive heuristic makes an initial estimate (based on a default time). If tasks could be successfully finished in this time, the estimated runtime will be the runtime of the longest task. If the requested time is not enough to run tasks, the estimated task runtime will be the requested time multiplied by an integer factor (abrupt decision). Our inspiration to enlarge or shrink requested time is based on TCP congestion window ideas, which in bad situations make abrupt decisions and in good ones is careful.

Based on the estimated time to run a task, the heuristic sweeps the requests queue choosing the best (greatest throughput) set of possible requests in a greedy manner. An initial set with a maximum number of pending requests allowed is created with the first possible requests. After that, if a new possible request could improve the throughput, a previous chosen request is discarded and the new request is inserted into the set of chosen requests. If the chosen set provides more processors than the number of tasks, requests in the chosen set will be issued with fewer processors. The chosen set is requested and the process can be repeated if the requests were not enough to run all the tasks.

```
$freeSlots \leftarrow getFreeSlots(getResourceQueue())$;
$allowedRequests \leftarrow maxPendingRequests() - actualPendingRequests()$;
for $i = 1$ to allowedRequests do
    $freeSlot \leftarrow getNext(freeSlots)$;
    $Chosen \leftarrow Chosen \bigcup freeSlot$;
end
foreach $freeSlot$ in $freeSlots$ do
    if $freeSlot$ improves Chosen throughput then
        $worstRequest \leftarrow getWorstRequest(Chosen)$;
        $Chosen \leftarrow Chosen - \{worstRequest\}$;
        $Chosen \leftarrow Chosen \bigcup freeSlot$;
    end
end
$optimize(Chosen, numRemainingTasks)$;
$issueRequests(Chosen)$;
```

Algorithm 2: Adaptive heuristic.

4.4. *Strategy evaluation*

We have analyzed the presented heuristics via simulations. Our simulator is based on the model depicted in Fig. 2. In order to ease the analysis of the adaptive heuristic behavior, there is only one space-shared resource available to a grid broker. We used *conservative backfilling* as an idealized scheduler heuristic, as mentioned in Section 2.

In order to represent the user's requests from local users, we applied real supercomputer workloads as input for simulations. They are traces of real machines and were obtained from the Parallel Workload archive [34]. We filtered out jobs with missing request times. The workloads used are described in Table 1.

Unfortunately, grid workloads availability is not the same as supercomputer workloads. In fact, grid workloads are not available and the current state of practice utilizes supercomputer workloads as grid workloads. Besides, this is not applicable in the case studied here because these traces do not provide the information necessary (e.g. there is no way to know how many tasks were executed). Therefore, we decided to use a synthetic grid workload, which creates a large set of combinations in order to cover several possibilities. In our model, a job can vary in the number of tasks, task mean execution time, task heterogeneity (1x, 2x, 4x) and submission time. Two jobs are of the same type if they have exactly the same values for the number of tasks, task mean execution time and task heterogeneity. The task heterogeneity of 1x (homogeneous) means that all tasks run in the same time, 2x obeys a uniform distribution $U(mean/2, 3mean/2)$ and 4x is $U(mean/4, 7mean/4)$. Table 2 summarizes the parameters to generate grid jobs, rendering 36 possible combinations for job types. The submission time was a random number that could assume any time in supercomputer trace interval with the same probability.

The value used as the maximum requested time was 64 800 s (the lowest among the greatest values found in supercomputer workloads used). The initial time value to run a task was one hour (3600 s). We have run simulations where the limit for *maxPR* was 1, 2, 3, 4, 5 and 6 (the maximum value among sites considered [35]).

Therefore, the number of possible scenarios is: 36 types of job × 6 values for maxPR × 3 *workloads* of space-shared resources × 2 heuristics (static and adaptive) = 1296. Note that simulations are independent of each other, i.e. to every job there is a new simulation that does not consider previous information (e.g. adapted requested time).

The static heuristic is used as a baseline to adaptive heuristic performance evaluation. Thus, we present results as the speedup of adaptive heuristic over static heuristic; that is, the turn-around time obtained with the static heuristic divided by the turn-around time of the same job obtained with the adaptive heuristic. Thus, values greater than 1 indicate that the adaptive performed better and values smaller than 1 showed otherwise. Each point shown in the graphics is the speedup of the mean value for job execution time for simulations of at least 100 jobs with the same type.[1] Therefore, each point is related to one job type.

---

[1] The number of simulations was defined in order to provide a confidence level of 95% with an error less than 5% — based on procedure described in [36].

Table 1
Used workloads

Trace	System	Number of processors	Number of requests	Offered load (%)	Period
SDSC SP2	San Diego Supercomputer Center SP2	128	73 496	72	April/1998–December/2000
SDSC BlueHorizon	San Diego Supercomputer Center BlueHorizon	1152	250 440	73	April/2000–December/2000
CTC SP2	Cornell Theory Center SP2	512	79 302	54	July/1996–July/1997

Table 2
Possible values for each job parameter

Heterogeneity	1x$U(mean, mean)$, 2x$U(mean/2, 3mean/2)$ and 4x$U(mean/4, 7mean/4)$
Task mean execution time	100, 1000 and 10 000 s
Number of tasks per job	100, 1000, 10 000 and 100 000

Table 3
Average *Speedup* of adaptive heuristic over static heuristic

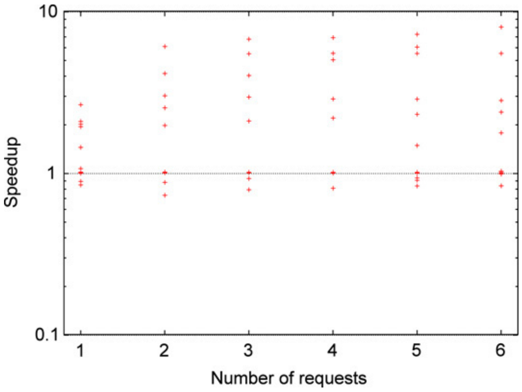
Workload	Average speedup
SDSC SP2	2.05
SDSC BlueHorizon	2.37
CTC SP2	14.74

The adaptive heuristic obtained better job execution time for most cases. The main difference in the results is due to the change of workloads. Table 3 summarizes the average *speedup* of adaptive heuristic over static heuristic for each *workload*.

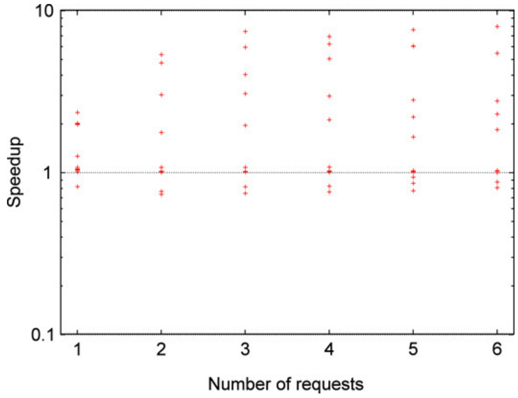
The graphics in Fig. 3 show the speedup between static and adaptive heuristics for all jobs utilizing the SDSC SP2 workload. From Fig. 3, it is easy to see that the adaptive shows better results in most cases (speedup for most cases is greater than 1). Moreover, the differences between Fig. 3(a) (homogeneous), (b) (heterogeneity 2x) and (c) (heterogeneity 4x) suggest that heterogeneity almost does not impact the results. Results in detail from SDSC BlueHorizon were omitted, since they are very similar to those from SDSC SP2.

Despite the fact that global results are better for adaptive heuristics, adaptive does worse for jobs with long task mean time (10 000 s) and few tasks (100 or 1000). Fig. 4 shows the speedup for such cases. These jobs render worse results to adaptive in comparison to static because the first requests issued by the adaptive heuristic cannot finish a task (which is very large); that is, static heuristic crafts "useful" requests (i.e. requests that can run at least a task) before "useful" requests from adaptive heuristic. The adaptive heuristic loses some requests during the process to estimate task runtime before making "useful" requests to a space-shared resource scheduler. It means that the learning process (generate a request, wait in queue, wait for execution, make a new request and repeat such cycle many times before making a good estimate) is longer than the necessary time to requests from the static heuristic finish.

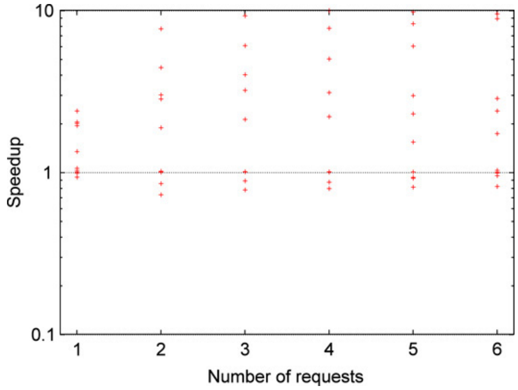
The biggest jobs (10 000 or 100 000 tasks of 1000 s or 10 000 s) present results of quite similar turn-around times (see Fig. 5); the speedup rates are around one. This happens because the needed time to estimate the task time is small in relation to the total job time and the requests crafted by the static heuristic already provide good throughput. Indeed, the requests produced



(a) Homogeneous.



(b) Heterogeneity 2x.



(c) Heterogeneity 4x.

Fig. 3. Grid jobs speedup of adaptive heuristic over static heuristic for SDSC SP2 workload.

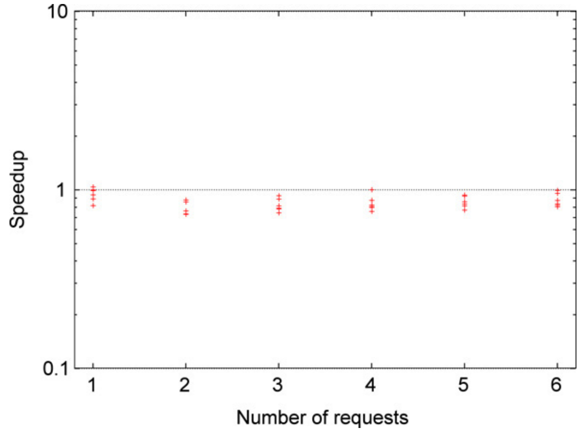


Fig. 4. Job speedup for cases with few tasks of long duration.

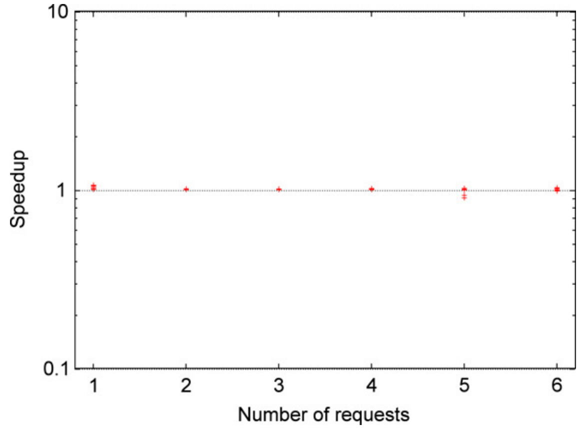


Fig. 5. Speedup for biggest jobs.

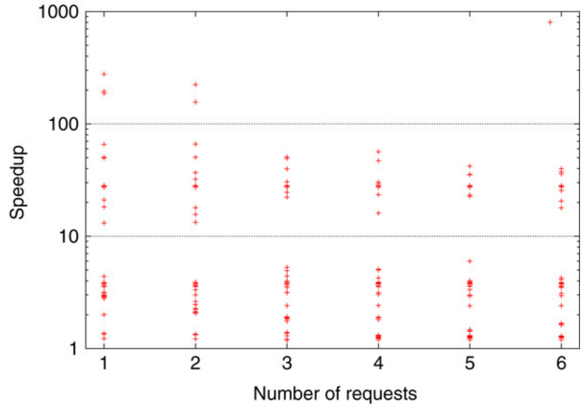


Fig. 6. Grid jobs speedup of adaptive heuristic over static heuristic for CTC workload.

by the adaptive heuristic are similar to those of the static one after the learning phase.

Fig. 6 shows the same results for the CTC SP2 workload. As heterogeneity does not render noticeable differences, we decided to present all workloads in only one figure. The adaptive heuristic performance was better in all cases (every point is greater than 1 in Fig. 6). The main difference is that CTC's load is smaller than that of the SDSC SP2. This implies more opportunities of backfilling to adaptive heuristic requests as they ask for fewer processors while static requests keep going to the end of the queue (they ask the maximum number of processors).

Another fact that to load explains this difference is the impact of the learning cycle performed by adaptive solution. In the cases in which the load is high, the time consumed on each iteration (mainly the waiting in queue) makes the learning process much longer.

In order to reinforce the analysis of load impacts, we artificially increased the CTC SP2 offered load to 78% by multiplying the submission time by a factor of 0.7. The jobs behaved similarly to the SDSC SP2 and SDSC BlueHorizon cases with a mean speedup of 1.83.

5. Strategies comparison

In this section we compare both strategies, analyze the results obtained from the experiments and provide some insights about situations where each strategy is better applied. We also describe a scenario that can benefit from utilization of both strategies.

From the experiments discussed in Section 3 and presented in [8], we can observe that the Transparent Resource Allocation strategy provides a reasonable performance for grid users when the execution time of each task is not large and the space-shared resources have medium to low utilization. It is even possible that long duration tasks could be executed in an acceptable time in space-shared resources with low utilization: what can happen in some periods, such as holidays, weekends and vacations. The main goal here is minimum influence to local utilization.

In spite of loss of performance when executing grid tasks, this strategy is well suited to use in sites where local users need higher priority in resource access, because resources will always be available to such users. We believe that this is acceptable because, commonly, grid users can obtain resources in more than one site, while local users of space-shared resources (typically, running high performance computing applications) usually depend on it efficiently to execute their applications, because, in general, tightly-coupled applications cannot be split efficiently among several sites. As intrusiveness caused by the Transparent Allocation Strategy is almost negligible, this strategy may motivate administrators to donate resources to the grid.

On the other hand, the Transparent Allocation Strategy will generate, to the grid user point of view, more faults, because resources can be pre-empted on behalf of local users any time; thus, the grid application can be aborted at any time, and it adds more costs (related to rescheduling) to the grid user application. However, experiments show that, depending on the characteristics of the application, overheads added by this strategy are acceptable. Moreover, one must remember that these resources would be wasted otherwise. Enabling the grid user to deliver speed-up from them is a definite plus.

Regarding the Explicit Allocation Strategy, experiments presented in Section 4 show that it is a useful strategy to be applied even in long duration tasks or in sites with high load. Nevertheless, both the Transparent Allocation Strategy and Explicit Allocation Strategy can be used efficiently in sites with low and medium loads. The Explicit Allocation Strategy is a costly strategy, as it should reserve resources to grid users. In such a strategy, grid users are deemed as local users, and therefore must be accepted by the local administrator as such. Therefore, we expect a given user to be able to use the Explicit Allocation Strategy on a smaller number of sites than she can reach via the Transparent Allocation Strategy. The best effort, non-intrusive characteristics of the Transparent Allocation Strategy encourage system administrators to make their resources more widely available. However, the utilization of resources by unknown users (from the grid) can also be a problem to the site administrator due to accounting and security.

Evaluating the intrusion of donating the resources and guarantees given by both strategies, we can distinguish the approximate cost in using the resources that can be applied to the strategies. Resources obtained using the Transparent Allocation Strategy are cheaper, since these resources would go idle otherwise. On the other hand, the cost of resources using the Explicit Allocation Strategy are more expensive, since there is an explicit reservation of resources that could be used by local users that “pay” for this privileged access.

We envisage a mixed utilization of both strategies using an economy model. The decision as to what strategy to use to request resources is based on the cost of the resources and the time available to obtain the results from the application. In this economy model, users need to “pay” for resources access and to specify a policy in which a grid scheduler will scavenge resources. This policy will decide whether the scheduler will minimize the spending, or accelerate execution completion, or even, try to execute the application based on a specific deadline with the minimum cost.

6. Conclusions and future work

Grid computing has been shown as an important tool to both science and industry in order to have access to more computational resources. These resources can be scientific instruments, storage, network bandwidth and processors. Processors used in grids can vary from idle workstations to space-shared resources, such as clusters of workstations or supercomputers.

This work presented the Explicit Allocation Strategy, which consists in deploying a heuristic to make a smart use of space-shared resources, granting to grid users access to an amount of resources as soon as possible. This strategy is a counterpart of our previous work, called the Transparent Allocation Strategy, which consists in donating to the grid resources while they are not in use by any local cluster users, preempting resources from the grid when they are requested by local users.

An important issue concerning the Transparent Allocation Strategy is the efficient fault tolerance support. This issue is critical, since the strategy aborts grid tasks being executed in nodes requested for local utilization. Since we focus on BoT applications, the integrity of the application is not affected when a task is aborted. The aborted tasks are inserted again in the set of tasks to be executed and resubmitted when processors become available. The main drawback of this approach is that the application or grid middleware must take explicit care of these faults.

A new type of resource scheduler, called Site Resource Scheduler (SRS), was introduced in [31]. It represents the site resources in the grid, making them available to higher Grid Schedulers, managing access rights and resources utilization. In such an approach, users ask (one or more) SRS about site capability. Based on the answers from the SRS site, users can divide their jobs among sites, delegating for each site an amount of tasks proportional to the site’s declared capacity. Afterwards, users can probe the status of the application, in order to identify whether the application had already finished or is taking too much time to be completed. By the application completion, the grid user can retrieve results generated by the application.

The Explicit Allocation Strategy can exploit the well-researched area of eager schedulers without modification as the allocation was opportunistic. The results show that it is possible to use the resources and the behavior of the heuristics: (i) the naive, static heuristic and (ii) adaptive heuristic. The adaptive heuristic performs better in most cases. Such a solution was also implemented for OurGrid [9].

In spite of being two different approaches to the same problem, the strategies are actually complementary: it is possible to deploy both techniques, allowing some users to use explicit allocation (due to the fact that those users are local users, or because the user payed to do that, or simply because the grid user already has an account for that space-shared resource), while others can only obtain transparent allocation. In fact, allowing different policies to different users can encourage systems administrators to deploy resources to the grid, increasing grid communities and contributing to the advancement of science. In future work, we intend to explore the economic incentives for site administrators to provide resources for the grid (via either or both strategies), as well as to determine whether and in which conditions we can support tightly coupled applications.

References

[1] W. Sudholt, K.K. Baldridge, D. Abramson, C. Enticott, S. Garic, C. Kondric, D. Nguyen, Application of grid computing to parameter sweeps and optimizations in molecular modeling, Future Generation Computer Systems 21 (1) (2005) 27–35.

[2] M. Cannataro, D. Talia, P. Trunfio, Distributed data mining on the grid, Future Generation Computer Systems 18 (8) (2002) 1101–1112.

[3] S. Smallen, W. Cirne, F. Berman, S. Young, M. Ellisman, J. Frey, R. Wolski, M.-H. Su, C. Kesselman, Combining workstations and supercomputers to support grid applications: The parallel tomography experience, in: HCW’00: Proceedings of the 9th Heterogeneous Computing Workshop, IEEE Computer Society, Washington, DC, USA, 2000, p. 241.

[4] J.R. Stiles, J. Thomas M. Bartol, E.E. Salpeter, M.M. Salpeter, Monte carlo simulation of neuro-transmitter release using mcell, a general simulator of cellular physiological processes, in: CNS’97: Proceedings of the Sixth Annual Conference on Computational Neuroscience: Trends in Research, 1998, Plenum Press, New York, NY, USA, 1998, pp. 279–284.

[5] D. Paranhos, W. Cirne, F. Brasileiro, Trading cycles for information: Using replication to schedule Bag-of-Tasks applications on computational grids, in: Proceedings of Euro-Par 2003: International Conference on Parallel and Distributed Computing, 2003, pp. 169–180.

[6] E. Santos-Neto, W. Cirne, F. Brasileiro, A. Lima, Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids, in: D.G. Feitelson, L. Rudolph (Eds.), Proceedings of 10th Workshop Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 3277, Springer-Verlag, 2004, pp. 210–232.

[7] N. Fujimoto, K. Hagihara, Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid, in: Proceedings of ICPP'2003 — 32th International Conference on Parallel Processing, 2003, pp. 391–398.

[8] M.A.S. Netto, et al., Transparent resource allocation to exploit idle cluster nodes in computational grids, in: Proceedings of the 1st International Conference on e-Science and Grid Computing, IEEE Computer Society Press, Melbourne, 2005, pp. 238–245.

[9] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, M. Mowbray, Labs of the World, Unite!!!, Journal of Grid Computing 4 (3) (2006) 225–246.

[10] D. Lifka, The ANL/IBM SP scheduling system, in: D.G. Feitelson, L. Rudolph (Eds.), Proceedings of 1st Workshop Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 949, Springer-Verlag, 1995, pp. 295–303.

[11] A. Bayucan, Portable Batch System Administration Guide, Veridian System.

[12] M.A.S. Netto, C.A.F. De Rose, CRONO: A configurable and easy to maintain resource manager optimized for small and mid-size GNU/Linux cluster, in: Proceedings of the 2003 International Conference on Parallel Processing, IEEE Computer Society Press, Kaohsiung, 2003, pp. 555–562.

[13] D.B. Jackson, Q. Snell, M.J. Clement, Core algorithms of the Maui scheduler, in: Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing, 2001, pp. 87–102.

[14] D.G. Feitelson, A. Mu'alem, Utilization and predictability in scheduling the IBM SP2 with backfilling, in: Proceedings of 12th International Parallel Processing Symposium, 1998, pp. 542–546.

[15] A.W. Mu'alem, D.G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Transactions on Parallel & Distributed Systems 12 (6) (2001) 529–543.

[16] J. Nabrzyski, J.M. Schopf, J. Węglarz (Eds.), Ten Actions When Grid Scheduling, Kluwer Academic Publishers, Norwell, 2003, pp. 15–23 (Chapter 2).

[17] J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke, Condor-G: A computation management agent for multi-institutional grids, Journal of Cluster Computing 5 (3) (2002) 237–246.

[18] D. Abramson, J. Giddy, L. Kotler, High performance parametric modeling with nimrod/g: Killer application for the global grid? in: IPDPS'2000, IEEE Computer Society Press, Cancun, 2000, pp. 520–528.

[19] E. Huedo, R.S. Montero, I.M. llorente, Experiences on adaptative grid scheduling of parammeter sweep applications, in: Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE Computer Society Press, A Coruna, 2004, pp. 28–33.

[20] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F.A.B. da Silva, C.O. Barros, C. Silveira, Running Bag-of-Tasks applications on computational grids: The mygrid approach, in: Proceedings of ICPP'2003 — 32nd International Conference on Parallel Processing, IEEE Computer Society Press, Kaohsiung, 2003, pp. 407–416.

[21] I. Foster, C. Kasselman, The Globus project: A status report, in: Proceedings of the Seventh Heterogeneous Computing Workshop, IEEE Computer Society Press, Orlando, 1998, pp. 4–18.

[22] I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, The International Journal of Supercomputer Applications and High Performance Computing 11 (2) (1997) 115–128.

[23] M.A. Marsan, G. Balbo, G. Conte, Performance Models of Microprocessor Systems, MIT Press, 1987.

[24] R. Wolski, Experiences with predicting resource performance on-line in computational grid settings, SIGMETRICS Performance Evaluation Review 30 (4) (2003) 41–49.

[25] S.A. Jarvis, et al., Performance prediction and its use in parallel and distributed computing systems, Future Generation Computer Systems 22 (7) (2006) 745–754.

[26] K. Aida, Effect of job size characteristics on job scheduling performance, in: D.G. Feitelson, L. Rudolph (Eds.), Proceedings of 6th Workshop Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 1911, Springer-Verlag, 2000, pp. 1–17.

[27] W. Cirne, F. Berman, A comprehensive model of the supercomputer workload, in: Proceedings of WWC-4: IEEE 4th Annual Workshop on Workload Characterization, 2001, pp. 140–148.

[28] W. Cirne, F. Berman, When the Herd is smart: The aggregate behavior in the selection of job request, IEEE Transactions in Parallel and Distributed Systems 14 (2) (2003) 181–192.

[29] C.B. Lee, Y. Schwartzman, J. Hardy, A. Snavely, Are user runtime estimates inherently inaccurate? in: D.G. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Proceedings of 10th Workshop on Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 3277, Springer-Verlag, 2004, pp. 253–263.

[30] DCSC/SDU Supercluster Horseshoe Cluster Statistics. http://www.dcsc.sdu.dk/docs/load/fairshare_info.php.

[31] W. Cirne, F. Brasileiro, L. Costa, D. Paranhos, E. Santos-Neto, N. Andrade, C.D. Rose, T. Ferreto, M. Mowbray, R. Scheer, J. Jornada, Scheduling in Bag-of-Task grids: The PAUÁ case, in: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, IEEE Computer Society Press, Foz do Iguaçu, 2004, pp. 124–131.

[32] A. Downey, Using queue time predictions for processor allocation, in: D.G. Feitelson, L. Rudolph (Eds.), Proceedings of 3rd Workshop on Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 1291, Springer-Verlag, 1997, pp. 35–57.

[33] J. Pruyne, M. Livny, Parallel Processing on Dynamic Resources with Carmi, in: D.G. Feitelson, L. Rudolph (Eds.), Proceedings of 1st Workshop on Job Scheduling Strategies for Parallel Processing, in: Lecture Notes in Computer Science, vol. 949, Springer-Verlag, 1995, pp. 337–360.

[34] Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[35] NPACI Users Guide: Blue Horizon. http://www.npaci.edu/Horizon/, June 2004.

[36] R. Jain, The Art of Computer Systems Performance Analysis, 1st ed., Wiley Interscience, John Wiley & Sons, Inc., New York, NY, 1991.

César A.F. De Rose is an Associate Professor in the Computer Science Department at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil. His primary research interests are parallel and distributed computing and parallel architectures. He is currently conducting research on a variety of topics applied to clusters and grids, including resource management, resource monitoring, distributed allocation strategies and virtualization.

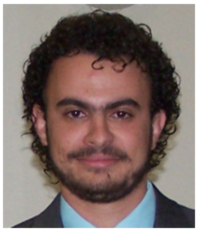
Dr. De Rose received his doctoral degree in Computer Science from the University Karlsruhe, Germany, in 1998. He currently leads the Research Center in High Performance Computing (CPAD-PUCRS/HP) at PUCRS.

Tiago Ferreto is an Assistant Professor in the Computer Science Department at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil. He received his B.E. and M.S. degrees from the Computer Science Department, PUCRS, Brazil, and is currently a Ph.D. candidate in the Computer Science Department at PUCRS. His primary academic research interests are resource management, grid computing and virtualization.

Rodrigo N. Calheiros is a Ph.D. student in Computer Science at Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil. He received his Master's degree from the same university in 2006. His research interests include resource management and scheduling, application of virtualization, and programing in grid computing and distributed systems.

Walfredo Cirne is with Google's infrastrucutre group in California, USA. He is on leave of his faculty position at the Computer Science Department of the Universidade Federal de Campina Grande, in Brazil. Dr. Cirne holds a Ph.D. from the University of California San Diego, in the USA. Since 1997, his research focuses on Distributed Systems and Resource Management, having led the OurGrid project from 2001 to 2006. Further information and publications of Dr. Cirne can be found at http://walfredo.dsc.ufcg.edu.br/index_en.html.

Lauro B. Costa is a research assistant and developer in OurGrid project at LSD (Distributed Systems Lab) of UFCG (Federal University of Campina Grande) in Brazil. He joined LSD in 2001 as an undergraduate student, he received BS degree in Computer Science from UFCG and he's been contributing as research assistant since the beginning of 2006, when he concluded his Master thesis in Computer Science from UFCG. Currently, his work is on folksonomy and resource/service discovery for grids via P2P GIS.



Daniel Fireman is a M.Sc. student at UFCG (Federal University of Minas Gerais) in Brazil and he is researching on dynamic reconfigurable distributed systems. Previously, he worked at LSD (Distributed Systems Lab) of UFCG (Federal University of Campina Grande). He had worked at LSD as part of OurGrid team until 2006, when he received his B.S. degree in Computer Science from UFCG.