A Fair Decentralized Scheduler for Bag-of-tasks Applications on Desktop Grids

Javier Celaya[1] and Loris Marchal[2]

February 2010

1:Aragón Institute of Engineering Research (I3A)
Dept. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, Zaragoza, Spain
`jcelaya@unizar.es`

2:CNRS & University of Lyon
LIP (ENS-Lyon-CNRS-INRIA-UCBL)
Lyon, France
`loris.marchal@ens-lyon.fr`

LIP Research Report RR-LIP 2010-07

Abstract

Desktop Grids have become very popular nowadays, with projects that include hundred of thousands computers. Desktop grid scheduling faces two challenges. First, the platform is volatile, since users may reclaim their computer at any time, which makes centralized schedulers inappropriate. Second, desktop grids are likely to be shared among several users, thus we must be particularly careful to ensure a fair sharing of the resources.

In this paper, we propose a decentralized scheduler for bag-of-tasks applications on desktop grids, which ensures a fair and efficient use of the resources. It aims to provide a similar share of the platform to every application by minimizing their maximum stretch, using completely decentralized algorithms and protocols. After presenting our algorithms, we evaluate them through extensive simulations. We compare our solution to already existing centralized ones under similar conditions, and show that its performance is close to the best centralized algorithms.

1 Introduction

Taking advantage of unused cycles of networked computers has emerged as a cheap alternative to expensive computing infrastructures. Due to the increasing number of personal desktop computers connected to the Internet, a tremendous computing power is potentially at hand. Desktop Grids gathering some of these machines have become widespread thanks to popular projects, such as Seti@home [19] or Folding@home [20]. Nowadays, projects as World Community Grid [23] include hundreds of thousands of available computers.

At a smaller scale, software solutions have been proposed to harness idle cycles of machines in the local area network scale [16]. This allows to use idle desktop computers located in places like a computer science laboratory or a company, for processing compute-intensive applications.

The key characteristic of these platforms that strongly limits their performance is volatility: a machine can be reclaimed by its owner at any time, and thus disappear from the pool of available resources [13]. This motivates the use of a robust distributed architecture to manage resources, and the adaptation of peer-to-peer systems to computing grids is natural [8, 9].

Target applications of these desktop grids are typically embarrassingly parallel. In the context of computing Grids, a common model for such applications is the bag-of-tasks: each application is then described as a set of similar tasks, i.e. which have a common data file size and computing demand [7, 21].

Due to the distributed nature of desktop grids, several concurrent applications, originating from different users, are likely to compete for the resources. Traditionally, schedulers of desktop grids aims at minimizing the overall completion time of an application. However, in a multi-application setting, it is important to maintain some fairness between users: we do not want to favor an application with a large number of small jobs compared to another application with fewer larger jobs. Similarly, if applications can be submitted at different entry points of the distributed system, we do not want that the location of the user impacts its experienced running time. To discourage users tampering with their application to get better performance, we must provide a scheduler that gives a fair share of the available resources to each user. Similar problems have been addressed in computing Grids [7, 14]. However, these schedulers are centralized, and assume perfectly updated information on the whole platform. In the context of desktop grid, a scheduler needs to be decentralized and rely only on local information.

In this paper, we propose and evaluate a decentralized scheduler for processing bag-of-tasks applications on desktop grids. Our study relies on previous work which proposes a peer-to-peer architecture to distribute tasks provided with deadlines [9]. We also build upon a previous study on a centralized scheduler for multiple bag-of-tasks applications on a heterogeneous platform [7].

2 Related work

2.1 Desktop grid and scheduling

Desktop grids are now widespread, and many platform management software is available [11]. Among others, BOINC [4] is probably the most common, and uses a classical client/server architecture. Other types of architecture are also proposed, some of them inspired by peer-to-peer systems [8].

To cope with node volatility, several mechanisms have been proposed, such as checkpointing and job migration [3, 5, 24]. These mechanisms allow to efficiently manage computing resources that are likely to be reclaimed by their owners at any time. Similarly, Kondo et al. [13] emphasize the need for resource selection when processing short-lived task-parallel application on desktop grids. These studies are complementary to our work, which focuses on how to share the available resources among several users.

The common scheduling policy in desktop grids is usually FCFS (First Come, First Served), but more complex strategies have also been proposed. In particular, Al-Azzoni et al. [2] propose to use linear programming to compute a mapping of applications to the platform. However, reactivity is achieved at the price of solving a linear program at each change of the platform, which makes it not very suited to volatile platforms. Besides, only a centralized scheduler can gather the whole information on the optimization problem.

2.2 Scheduling multiple applications

In the context of classical computing Grids, the problem of scheduling multiple applications have already been studied. As far as fairness is concerned, the most suited metric seems to be the maximum stretch, or slowdown [14]. The stretch of an application is defined as the ratio of its response time under the concurrent scheduling policy over its response time in dedicated mode, i.e., when it is the only application executed on the platform. The objective is then to minimize the maximum stretch of any application, thereby enforcing a fair trade-off between all applications.

Previously, we have studied the minimization of maximum stretch for concurrent applications in a centralized settings [7]. In particular, our study shows that interleaving tasks of several concurrent bag-of-tasks applications allows to reach a better performance than scheduling each application after the other.

2.3 Distributed scheduling

Distributed scheduling has been widely studied in the context of real-time systems, when tasks have deadline constraints. Among others, Ramamritham et al. [18] propose several decentralized heuristics to schedule real-time applications in a distributed environments. More recently, Modi et al. [17] present a distributed algorithm to solve general constraint optimization problem with a guaranteed convergence.

However, these studies are dedicated to tasks with deadlines, and stable environments where complex distributed algorithms can converge. In our large-scale and fault-prone system, we cannot hope to reach optimality, and we will rather design fault-tolerant heuristics inspired by peer-to-peer techniques.

Closer to our problem, Viswanathan et al. [22] proposed a distributed scheduling strategy for computing grid. However, a centralized entity is used to gather the information on the platform and the applications.

In earlier work, we have compared centralized and decentralized strategies for scheduling bag-of-tasks applications in computing grids [6], however in this study, applications were supposed to be available at the same time on a given master node, whereas applications are likely to be issued at any time and any place in a desktop grid.

3 Problem description

In this section, we formally define the problem we target. Our goal is to design a fully decentralized scheduling architecture for bag-of-tasks applications, oriented to desktop grid platforms. Our main objective while scheduling tasks of concurrent applications is to ensure fairness among users.

3.1 Application model

Each application A_i consists of a set of n_i tasks with computing demand a_i, measured in millions of flops. Let $w_i = n_i a_i$ be the overall computing size of the application. Each application A_i has a release time r_i, corresponding to the time when the request for its processing is issued, and a finish time C_i, when the last task of this application terminates.

When scheduling multiple applications, as far as fairness is concerned, the most suited metric seems to be the maximum stretch, or slowdown [14]. The stretch of an application is defined as the ratio of its response time under the concurrent scheduling policy $(C_i - r_i)$ over its response time in dedicated mode, i.e., when it is the only application executed on the platform. The objective is then to minimize the maximum stretch of any application, thereby enforcing a fair trade-off between all applications.

In a distributed context, it is hard to evaluate the response time of an application in dedicated platform (needed to compute the stretch), since we do not even know the overall number of nodes. If we would know what is the aggregated computing speed s_{agg} of the whole platform, then we could approximate this response time as w_i/s_{agg}, assuming a perfect distribution of the application on the computing node. The stretch for application A_i would then be $(C_i - r_i)/(w_i/s_{\text{agg}})$.

In practice, we do not known the value of s_{agg}, but we assume that it does not vary much, and that its variations should not be taken into account when computing the slowdown of each application. Thus, we assume that the aggregated speed has a constant value, and we approximate the stretch with $(C_i - r_i)/w_i$.

3.2 Platform model

The platform model which we are using is inherited from the framework described in [9]. Nodes are organized in a network overlay based on a balanced binary tree, where every leaf node is a processing node, and every internal node is a routing node. The actual implementation of such overlay is not detailed in this paper, since significant work is already done on this subject [1, 12, 15]. These solutions propose tree-based peer-to-peer overlays with good scalability and fault-tolerance properties. Each machine taking part of the computation acts simultaneously as a processing node and a routing node of the overlay. The computational speed of a computing (leaf) node of the overlay is denoted by s_u, measured in millions of flops per second.

3.3 Scheduling on the overlay

We use the tree structure of the overlay both for gathering information on the platform and for scheduling. The information of the platform availability is aggregated from the leaf nodes to the root node, as detailed below in Section 5.

When an application is released, the corresponding request, containing all necessary information on the application, is received by some machine the system. The routing node of this machine processes the request based on the information it holds on the platform. The routing node can either decide to process the application locally in its own subtree, if the application is small enough and will not cause a large load imbalance, or it can decide to forward the application to its father in the overlay, which now faces the same choice. When finally, a node (possibly the root node) takes the decision to schedule the application in its subtree, it splits the application and allocates a number of its tasks to each of its children. Then, the children must take the same decision, until the tasks reach the leaf nodes. The leaf node inserts the incoming tasks into their task queue, and processes them.

In the following, we first present the local scheduling policy, used by the leaf node to order their task queue (Section 4). Then we explain how the availability of the platform is gathered along the tree (Section 5), and finally we describe the global scheduling policy (Section 6).

4 Local scheduler

Each execution node has a local scheduler which decides the processing order of tasks allocated to this node. The local scheduler has the same objective as the whole platform: minimizing the maximal stretch among all applications. We rely on a relation between stretch and deadlines:

$$S_i = \frac{d_i - r_i}{w_i} \implies d_i = r_i + S_i w_i \qquad (1)$$

Given a value S for this maximum stretch, we can thus compute a deadline for all the tasks of every application. Then, we can schedule all tasks using and Earliest Deadline First (EDF) policy, as detailed in Alkgorithm 1: if the deadlines are achievable, the EDF policy finds a suitable schedule. Finally, we apply a binary search to find the minimal possible value for the stretch: for a given stretch value, we compute the deadlines with the previous formula, and apply the EDF policy; if the deadlines are met, we start again with a smaller stretch values; if they are not met, we increase the stretch value. Algorithm 2 details this binary search.

Desktop Grid environments are particularly error prone. Thus, fault-tolerance is a required capacity of algorithms dedicated to such platforms. In the context of the local scheduling, if a failed node had any task in its queue, they are aborted. At this moment, we just resubmit tasks when they are detected to have failed. This affects the stretch of the applications involved since they last longer than expected. On the other hand, as it is empirically shown in Section 7, having some tasks from all or part of the applications resubmitted gives the scheduler the opportunity to recalculate their stretch and achieve more similar values between different applications.

Algorithm 1: Algorithm *meetDeadlines*(Q).

```
Input: $Q$ is a task queue.
Output: Whether all tasks in Q meet their deadlines.
$e$ = currentTime
Order tasks by non-decreasing deadlines: $d_1 \leq d_2 \leq \cdots$
forall tasks in $Q$ do
    $e = e + a_i/s_u$
    if $e > d_i$ then return false
return true
```

5 Platform's availability

In this section, we detail the process that gathers information about the state of the platform, and communicates it to distant nodes, so that each node can be able to efficiently schedule an application. The state of the platform is based on the availability of nodes to receive new tasks. Each computing node builds an *availability summary*, which states how many new tasks it can receive. This availability summary is then gathered by the routing node of the tree, until it reaches the root node. This availability summary is designed both to provide complete availability on the platform, and to induce a limited communication overhead.

5.1 Computing the availability of nodes

In order for the global scheduler to correctly allocate tasks to the platform, each computing node must provide an availability summary which described its capacity to process tasks from new applications. The availability of a node consists in the number of tasks of a new application that this node is able to process. Of course, this number of tasks both depends

Algorithm 2: Algorithm for function *maxStretch*(Q, ϵ).

```
Input: $Q$ is the task queue, $\epsilon$ is the error tolerance.
Output: Optimal maximum stretch that makes all applications meet deadlines.
$S_{\text{min}} = 0$, $S_{\text{max}} = 1$
for $i = 1$ to $|Q|$ do $d_i = r_i + S_{\text{max}} \cdot w_i$
Sort applications by increasing deadlines $d_i$.
while meetDeadlines($Q$) is false do
    $S_{\text{min}} = S_{\text{max}}$
    $S_{\text{max}} = 2S_{\text{max}}$
    for $i = 1$ to $|Q|$ do $d_i = r_i + S_{\text{max}} \cdot w_i$
while $S_{\text{max}} - S_{\text{min}} > \epsilon$ do
    $S_{\text{mid}} = (S_{\text{max}} + S_{\text{min}})/2$
    for $i = 1$ to $|Q|$ do $d_i = r_i + S_{\text{mid}} \cdot w_i$
    if meetDeadlines($Q$) then
        $S_{\text{max}} = S_{\text{mid}}$
    else
        $S_{\text{min}} = S_{\text{mid}}$
return $S_{\text{max}}$
```

on the target maximum stretch and on the application itself. Formally, for a given target stretch S, and assuming that the new application will be released at time r_{new}, is made of tasks of length a_{new}, and has total size w_{new}, we compute $n(S, r, w, a)$, the maximal number of tasks that the local computing node can handle locally.

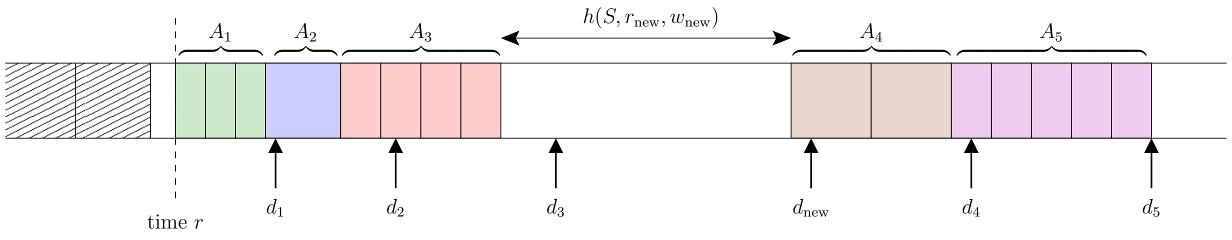


Figure 1: Example task queue. Tasks from applications A_1 through A_3 are processed as soon as possible, while applications A_4 and A_5 are processed as late as possible. The available computation in the gap between them is $h(S, r_{\text{new}}, w_{\text{new}})$.

We briefly describe the evaluation of this function through the example depicted in Figure 1. The complete algorithm to compute n is available in the companion research report [10]. Given a tuple $(S, r_{\text{new}}, w_{\text{new}}, a_{\text{new}})$, that is, assuming that a new application A_{new} with tasks of size a_{new}, and total size w_{new} will be released at time r_{new}, we have to evaluate the number of such tasks which can be processed while ensuring a stretch at most S. The first step is to construct the task queue that it expects to have at time r_{new}: it discard from its actual task queue the tasks that will be processed at time r (the shaded tasks in Figure 1). Then, we compute the deadline that each remaining application A_i would have with stretch S, using Equation (1). The deadline d_{new} of the new application A_{new} is also computed, and all deadlines are sorted in a non-decreasing order, since the tasks are going to be scheduled using an EDF policy. In the example of Figure 1, d_{new} lies between d_3 and d_4. The earliest starting time for the tasks of A_{new} is after tasks of A_1, A_2, and A_3 (that is all tasks with $d_i < d_k$) have been computed. Similarly, the latest completion time for the tasks of A_{new} must ensure that tasks of A_4 and A_5 (that is all tasks with $d_i > d_k$) will not miss their deadlines, and also that deadline d_{new} is not exceeded. This allows to compute the duration of the time slot that can be devoted to A_{new}, denoted by $h(S, r_{\text{new}}, w_{\text{new}})$. Then, the number of atomic tasks that can be processed by the node is given by:

$$n(S, r_{\text{new}}, w_{\text{new}}, a_{\text{new}}) = \left\lfloor \frac{h(S, r_{\text{new}}, w_{\text{new}})}{a_{\text{new}}} \right\rfloor \tag{2}$$

Algorithm 3 computes the function $h(S, r_{\text{new}}, w_{\text{new}})$. It first calculates the deadline d_i of every application in the queue with stretch S, and d_{new} for a potentially new application of parameters r_{new} and w_{new}. Then the queue is sorted in EDF order, and the algorithm first computes the latest starting time x_i of each application A_i such that no application A_j with $j \geq i$ misses its deadline. For any combination of parameters which makes any application miss its deadline, the function returns 0 (the stretch is not achievable). Then, the amount of computation (number of flops) available between r_{new} and d_{new} is calculated. For sake of simplicity, we assume that applications are ordered by non-decreasing value of d_i ($d_i \leq d_{i+1}$), and that the remaining number of tasks for application A_i is N_i^u.

Algorithm 3: Algorithm for function $h(S,\ r_{\text{new}},\ w_{\text{new}})$.

```
Input: $S$ is the desired stretch, $r_{\text{new}}$ is the release date of the new application, $w_{\text{new}}$
       is its size.
Output: number of flops available for the new application.
for $i = 1$ to $n$ do $d_i = r_i + S \cdot w_i$
$d_{\text{new}} = r_{\text{new}} + S \cdot w_{\text{new}}$
Order tasks by non-decreasing deadlines: $d_1 \leq d_2 \leq \cdots$
$x_n = d_n - N_n^u \cdot a_n / s_u$
for $i = n - 1$ to 1 do
    $x_i = \min(d_i, x_{i+1}) - N_i^u \cdot a_i / s_u$
if $x_1 <$ current_time then return 0 (at least one application misses its deadline)
Get $k$ so that $d_{k-1} < d_{\text{new}} \leq d_k$
$e_k = r_{\text{new}} + \sum_{i=1}^{k-1} N_i^u a_i / s_u$
return $(\min(d_{\text{new}}, x_k) - e_k) s_u$
```

Deadline constraints are checked with the use of x_1. If the first application is forced to start before the current time, then it means that one or more applications are missing their deadline with the selected stretch S. Then, the position of the new application in the queue is calculated. The result is the number of flops that can be executed between the moment at which the previous application is going to finish, and the deadline of the new application or the last moment at which next application must start, whichever comes first. Call e_k the moment at which application $k-1$ in the queue is expected to finish, it can be calculated by adding the remaining execution time of the $k-1$ first applications to r. Then, if the new application would be at position k in the queue we have:

$$h(S, r, w) = (\min(d_{\text{new}}, x_k) - e_k)s_u \tag{3}$$

5.2 Availability summary

One of the main part of routing nodes consists in dividing a set of tasks from a new application to send a subset of these tasks to each of its sub-branches (we recall that routing nodes are organized on a tree-based overlay). In order to decide how to split a set of tasks, the routing nodes must know how many tasks can be processed by each subtree rooted at each of its children, and how the stretch of the nodes in that subtree is going to be affected by the new application. This information is provided by the local schedulers through the availability function, and aggregated at each level of the tree as a *summary* of the availability of the nodes of each branch.

The availability function n provided by each execution node is not directly suited for the aggregation. For this reason, it is summarized in a four-dimensional matrix, called *availability summary* which contains samples of this function: each cell of the matrix, identified by a tuple $(S^{(i)}, r^{(j)}, w^{(k)}, a^{(l)})$, contains a conservative approximation of the function for these parameters. A routing node receives a similar matrix (for the same selected values of the parameters) from each of its children. In order to report a global availability summary to its father, it simply aggregates all the received matrices by adding them. By doing so, when a new application is released, the resulting matrix provides the number of tasks that can be sent to that node, with a guaranteed maximum stretch.

In order to extract the correct information from the availability matrix, routing nodes look for the cells associated to the parameters r_i, w_i, a_i contained in each request. However, not every combination of parameters can be taken into account when creating the availability matrix, due to space limitations. These parameters are discretized into a set of selected values. Thus, whenever a new application arrives, the cells used in the division process are those identified by the nearest values to the application parameters. What is more, since the availability summary contains only samples of the original function, we a priori do not know what happens between two selected values of the parameters. In order to better understand the behavior of the n function based on the availability summary, and to be able to get guaranteed interpolated availability between the selected values for the parameters, we carefully study the evolution of n for each parameter. This also helps us to decide which is the best selected values for parameters $S^{(i)}$, $r^{(j)}$, $w^{(k)}$ and $a^{(l)}$. In the rest of this section, we will abbreviate $n(S, r, w, a)$ by $n(S)$ when the other parameters (r, w and a) are fixed, and similarly for the h function defined in Section 4 or other parameters. Note that we study the evolution of the n function on a given computing node P_u (with speed s_u). Then, when the availability of several subtrees are aggregated into a single summary, we simply add n functions coming from each subtree. The properties on this function exhibited below concern its monotonicity for various parameters, so they are naturally conserved by the aggregation.

5.2.1 Evolution of n with task size a

From Equation 2, it is clear that $n(a)$, evolves as a descending staircase with increasing values of a. These steps can be calculated as:

$$n(a) = \begin{cases} i, & a \in \left(\frac{h(S,r,w)}{i+1}, \frac{h(S,r,w)}{i}\right] \ \forall i \in \mathbb{N} \\ 0, & a > h(S,r,w) \end{cases}$$

It can be seen that more precision is needed for smaller values of a. For this reason, the values $a^{(i)}$ will be taken from a geometric succession $a^{(i)} = b^i$. b can be empirically determined, but for now we will use $b = 2$. The execution nodes will eventually decide for which of these values they provide information, because it is useless to calculate the function n for tasks with a length under or over certain limits. For example, a task with $a_i = 2^{16}$ would need around one minute to execute on a node with $s_u = 2^{10}$, which given the nature of the platform seems to be a suitable minimum value.

5.2.2 Evolution of n with application size w

For the other three parameters, the evolution of function h will be studied first. First of all, we will prove that the larger the new application, the more time it will be devoted on node P_u, which is expressed by the following proposition:

Proposition 1. *$h(w)$ is monotonically non-decreasing.*

Proof. As it can be deduced from Equation (1), fixed r and S, the deadline of a new application is a linear function of w, let call it $d(w) = S \cdot w + r$. Due to the EDF ordering, applications are sorted by non-decreasing deadlines. Let us assume that applications are numbered such that $d_{i-1} < d_i \ \forall i$, then the new application will maintain its position k in the application queue as long as $d(w) \in (d_{k-1}, d_k]$, which means $w \in \left(\frac{d_{k-1}-r}{S}, \frac{d_k-r}{S}\right]$.

Given w so that $d(w) \in (d_{k-1}, d_k]$, from Equation (3) we have $h(w) = (\min(d(w), x_k) - e_k) \cdot s_u$. As w increases, the first situation is $d(w) < x_k$ and then $h(w)$ is a linear function of $d(w)$ with slope s_u, so it is a linear function of w with slope $S \cdot s_u$. Afterward, when $d(w) \geq x_k$, $h(w)$ has constant value $(x_k - e_k) \cdot s_u$. An example of both situations can be seen in Figure 2. In conclusion, $h(w)$ is monotonically non-decreasing in the interval $\left(\frac{d_{k-1}-r}{S}, \frac{d_k-r}{S}\right]$ for every k. In the special case where $k = 1$, d_0 makes no sense and can be replaced in the deduction by r. In the special case where the new application will be executed after the n applications in the queue, so $k = n+1$, d_{n+1} makes no sense and $x_{n+1} = \infty$, thus the function $h(w)$ is a linear function in the interval $\left(\frac{d_n-r}{S}, \infty\right]$ with slope $S \cdot s_u$.

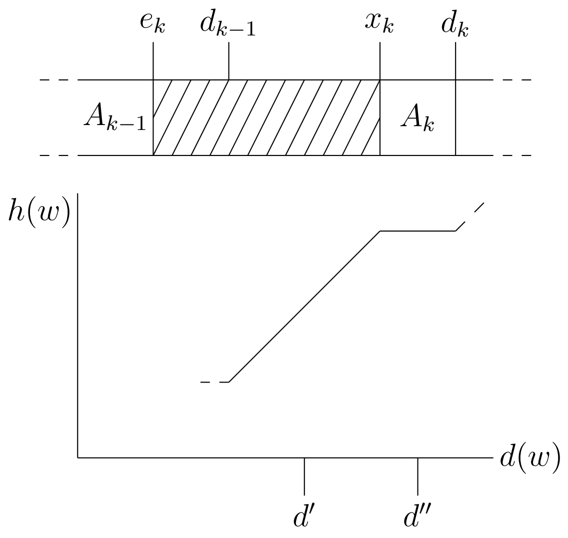


Figure 2: At $d(w) = d'$, $h(w) = (d(w) - e_k)s_u$. At $d(w) = d''$, $h(w) = (x_k - e_k)s_u$.

Finally, when $d(w)$ reaches the deadline d_k of some other application, we have $x_k < d_k$ and thus $x_k < d(w)$, so $h(w) = (x_k - e_k) \cdot s_u$, as explained before. Now, if we add and substract the size of application A_k:

$$h(w) = (x_k - e_k) \cdot s_u = \left(x_k + \frac{N_k^u a_k}{s_u} - e_k - \frac{N_k^u a_k}{s_u} \right) \cdot s_u$$

$e_k + \frac{N_k^u a_k}{s_u}$ trivially equals to e_{k+1}, and from definition of x_k in Algorithm 3 we have:

$$x_k = \min(d_k, x_{k+1}) - \frac{N_k^u a_k}{s_u} \;\Rightarrow\; x_k + \frac{N_k^u a_k}{s_u} = \min(d_k, x_{k+1})$$

To sum up, with $d(w) = d_k$,

$$\begin{aligned} h(w) &= (\min(d_k, x_{k+1}) - e_{k+1})s_u \\ &\leq (\min(d', x_{k+1}) - e_{k+1})s_u,\ \forall d' \in (d_k, d_{k+1}] \end{aligned}$$

So, the maximum value of $h(w)$ with w in interval k is lower than or equal to any value of $h(w)$ in interval $k+1$, and thus $h(w)$ is monotonically non-decreasing in all its domain. $\square$

Being $h(w)$ monotonically non-decreasing means that $n(w)$ is also non-decreasing. This result is important in order to compute the correct value of $n(S, r, w, a)$ for each cell of the matrix in an availability summary. For the cells with parameter $w = w^{(i)}$, the stored value must be the number of tasks that the node is available to execute for a new application with parameter $w \in [w^{(i)}, w^{(i+1)})$. Being the function $n(w)$ non-decreasing, this number can be obtained just by sampling the function with $w = w^{(i)}$.

For the actual values of $w^{(i)}$, the same geometric succession as for $a^{(i)}$ is used, since w_i is a multiple of a_i for all applications A_i. This also guarantees that the ratio between the actual value of w and the one used in the matrix is at most b.

5.2.3 Evolution of n with stretch S

For the stretch parameter, we derive similar results as for w:

Proposition 2. *$h(S)$ is monotonically non-decreasing.*

Proof. Modifying the value of S modifies the deadline of all the applications in a node, so the order of their execution may change. In fact, two applications will exchange their position in the queue when their deadlines become equal:

$$d_i = d_j \;\Leftrightarrow\; r_i + S_{i,j}w_i = r_j + S_{i,j}w_j \;\Leftrightarrow\; S_{i,j} = \frac{r_j - r_i}{w_i - w_j}$$

$S_{i,j}$ only makes sense when it is positive. If it is not, then applications i and j do never exchange positions. Assuming that applications are ordered by its release date, so that $r_i < r_j \Leftrightarrow i < j$, then it is trivial that $S_{i,j} > 0 \Leftrightarrow w_i > w_j$. That is, applications i and j exchange positions only if w_i is greater than w_j. When $S < S_{i,j}$ application A_i will execute before A_j, and when $S > S_{i,j}$ they will execute in reverse order.

Again, the available computation for the new application when it occupies position k in the queue is $h(S) = (\min(d(S), x_k(S)) - e_k)s_u$. It is trivial to see that, when two applications before position k exchange positions, e_k does not change, as it is the sum of the remaining time of the $k-1$ first applications in the queue. The same is true for x_k, it does not change when two applications exchange positions after position k. From Algorithm 3 we have:

$$x_k = \min(d_k, x_{k+1}) - \frac{N_k^u a_k}{s_u}$$

Supposed that application A_i is at position k and application A_j is at position $k+1$, when $S = S_{i,j}$ we have $d_i = d_j$ and thus x_k has the same value no matter in which order application A_i and A_j are:

$$\begin{aligned}
x_k &= \min(d_i, \min(d_j, x_{k+2}) - \frac{N_j^u a_j}{s_u}) - \frac{N_i^u a_i}{s_u} \\
&= \min(d_i, \min(d_i, x_{k+2}) - \frac{N_j^u a_j}{s_u}) - \frac{N_i^u a_i}{s_u} \\
&= \min(d_i, x_{k+2}) - \frac{N_j^u a_j}{s_u} - \frac{N_i^u a_i}{s_u} \\
&= \min(d_i, x_{k+2}) - \frac{N_i^u a_i}{s_u} - \frac{N_j^u a_j}{s_u} \\
&= \min(d_i, \min(d_i, x_{k+2}) - \frac{N_i^u a_i}{s_u}) - \frac{N_j^u a_j}{s_u} \\
&= \min(d_j, \min(d_i, x_{k+2}) - \frac{N_i^u a_i}{s_u}) - \frac{N_j^u a_j}{s_u}
\end{aligned}$$

So, the only values of the stretch which influence the available computation for the new application are:

$$S_i = \frac{r - r_i}{w_i - w}$$

When $S = 0$, the new application will be the last in the queue. As it increases, it will advance in the queue over each application with $w_i > w$ when it arrives at S_i. For simplicity in notation, we assume that $\forall i\, S_i > 0$ and $S_i > S_j \Leftrightarrow i < j$. Thus, when $S \in (S_k, S_{k-1}]$, the new application would be in the k position of the queue, with applications $A_1, \dots, A_{k-1}$ before it and $A_k, \dots, A_n$ after it.

When S is at the beginning of the interval, the new application would have just exchanged position with application A_k, so $d(S) \geq x_k(S)$ and $h(S) = (x_k(S) - e_k)s_u$. $x_k(S)$ is a piecewise function where each segment is a linear function which depends on $d_i, k \leq i \leq n$, so its slope is one of $w_i, k \leq i \leq n$. Thus, the minimum slope of $x_k(S)$ is $\min_{i=k} n w_i > w$. $x_k(S)$ grows faster than $d(S)$, which slope is w, so for certain value of S we have $d(S) < x_k(S)$, and then $h(S) = (d(S) - e_k)s_u = (r + Sw - e_k)s_u$ is a linear function of slope ws_u.

So function $h(S)$ when $S \in (S_k, S_{k-1}]$ consists of two linear segments, the first with slope greater than ws_u and the next with a slope ws_u. It is trivially continuous when $d(S) = x_k(S)$, so we conclude that $h(S)$ is monotonically increasing in interval $(S_k, S_{k-1}]$. When $S \in (0, S_n]$, the new application is the last one so x_k makes no sense and the slope of $h(S)$ is ws_u in all the interval. When $S > S_1$, the new application is not advancing in the queue any more, so $h(S)$ grows forever with slope ws_u once $d(S) < x_1(S)$. Also, it is worth noting that $h(S)$ is zero while $e_k > \min(d(S), x_k(S))$.

Now we study the situation when $S = S_k$. In that situation $d(S) = d_k(S)$, so $x_k(S) \leq d(S) \leq d(S')$, $\forall S' \in (S_k, S_{k-1}]$. Before the exchange, $h(S) = (\min(d(S), x_{k+1}(S)) - e_{k+1})s_u$. Again, if we add and substract the size of application A_k:

$$\begin{aligned}
h(S) &= (\min(d(S), x_{k+1}(S)) - e_{k+1})s_u \\
&= (\min(d_k(S), x_{k+1}(S)) - \frac{N_k^u a_k}{s_u} - e_{k+1} + \frac{N_k^u a_k}{s_u})s_u \\
&= (x_k(S) - e_k)s_u \\
&\leq (\min(d(S'), x_k(S')) - e_k)s_u,\ \forall S' \in (S_k, S_{k-1}]
\end{aligned}$$

So, the maximum value of $h(S)$ with S in interval k is lower than or equal to any value of $h(S)$ in interval $k - 1$, and thus $h(S)$ is monotonically non-decreasing in all its domain. $\square$

The maximum value of S can be arbitrarily large, because it may increase as new applications enter the system. It is only limited by the capacity of the queue at each node. Theoretical stretch is lower bounded by one, but using our approximation, its minimum value tends to zero. The selected set of values for $S^{(i)}$ is again a geometric succession, but this time b is a real number between 0 and 1. The cells of the matrix with parameter $S = S^{(i)}$ will provide the number of tasks for a new application that rises the stretch to a value in the interval $(S^{(i+1)}, S^{(i)}]$. Again, with Proposition 2 the value at each cell is simply calculated by sampling function n at $S = S^{(i)}$.

Unlike parameters a and w, for which the minimum and maximum values are fixed during the system runtime, the minimum and maximum values of S for which information is provided by execution nodes depends on the context. The minimum stretch will be the one for which any application in the system misses its deadline, and thus the function n returns 0. The total number of samples values for S will be fixed, which imposes the maximum value for $S^{(i)}$.

5.2.4 Evolution of n with release time r

Finally, we consider the evolution of function n with respect to parameter r. If we consider an empty queue at node P_u, we have:

$$h(r) = (d(r) - r) \cdot s_u = (S \cdot w + r - r) \cdot s_u = S \cdot w \cdot s_u$$

In an empty queue, the computation available to a new application does not depend on r. In any other situation, node P_u will not process more tasks for the new application than when it is totally free, so h will be lower or equal to $S \cdot w \cdot s_u$. In Figure 3, it can be seen that function h is not monotonic with respect to parameter r. In the first case, available computation increases as r reaches e_k. In the second case, when $d(r)$ reaches x_k, available computation will decrease as r increases.

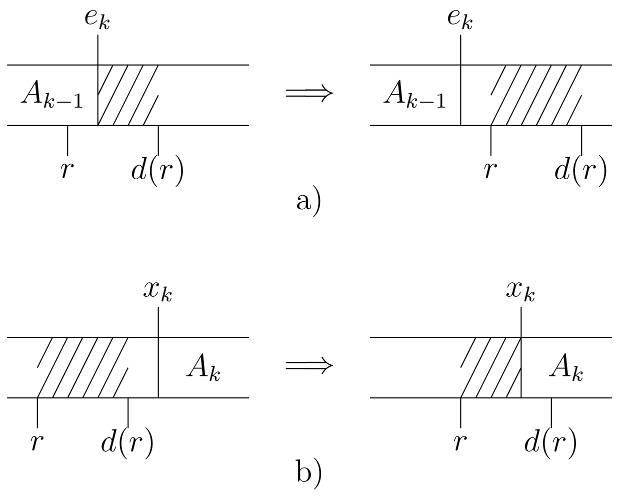


Figure 3: Two situations where (a) $h(r)$ increases as r increases and (b) $h(r)$ decreases as r increases. Thus $h(r)$ is not monotonic.

Alongside with this lack of monotony, we can question the interest of computing and aggregating availability summaries for many values of r: if r is too far in the future, it is very likely that an updated summary will be received in the meantime. Empirical results show that the best solution is to use just one value for parameter r, near in the future, and update the availability information periodically. By doing so, information is always up to date and no bandwidth is wasted because smaller availability summary matrices are sent. The right value for parameter r depends on the update period, which impacts the bandwidth consumption. In our test, we used a period of five minutes, and r is set to a time ahead of the current time so that the number of minutes is a multiple of five. In this way, two different matrices created at two different moments will have the same value for r if they differ in less than five minutes.

5.3 Updating the availability information

The availability information stored at each routing node of the tree may be updated for two reasons: (1) when the availability of a node in that branch changes, or (2) when a request is routed by that routing node.

Each time the availability of an execution node changes (for example when a new child arrives or leaves), it creates a new summary and sends it to its father node. The father will aggregate it with the ones coming from its other children and report the result to the next ancestor, until the root node is reached. If this update is performed in such a reactive way, each change in the leaves would lead to a message going up in the tree, which would quickly flood the upper levels of the tree with updates. To avoid this situation, an update rate limitation is set. When update messages are being sent at higher rates, they are discarded in favor of the newer messages.

A node must update the availability information of its children when it allocates some tasks to them, in order to avoid routing the next request to the same execution nodes. However, since the summary reports the availability of a whole subtree, it is difficult to predict the impact of allocated tasks. We adopt here a conservative approach, so that the summary matrix always contains a *lower bound* on the number of tasks the subtree is able to compute, for given values of the parameters. Assume we have just send N tasks from a new application with task size a_{new}. Then, we first subtract N to all cells with similar a value:

$$n(S, r, w, a_{\text{new}}) \leftarrow n(S, r, w, a_{\text{new}}) - N$$

All other cells are updated in a similar way, to account for the compute time of the new tasks:

$$n(S, r, w, a_{\text{new}}) \leftarrow n(S, r, w, a) - \left\lceil \frac{N \cdot a}{a_{\text{new}}} \right\rceil$$

This allows us to (roughly) estimate the new occupation of the subtree, before a real summary update is received.

6 Global scheduler

The global scheduling policy strongly relies on the availability information which is aggregated in the tree using the mechanism described in the previous section. The routing nodes perform the functionality of the global scheduler in a decentralized fashion: they receive an application allocation request at r_{new}, which contains the values N_{new}, a_{new} and w_{new} for a new application A_{new}, and route it throughout the tree to the execution nodes, trying to maintain the global maximum stretch as low as possible.

When a branch nodes receives a request for the allocation of a new application, it uses the availability summaries of its children to calculate the minimum stretch that can be achieved by using the resources in its own subtree, using a binary search among the stretch samples, as detailed in Algorithm 4. Specifically, the algorithm looks for the minimum sample value $S^{(i)}$ such that its own availability summary guarantees that all N_{new} tasks of the new application can be processed locally with stretch $S^{(i)}$.

Algorithm 4: Algorithm to compute the minimum local stretch

```
Input: availability summary $n^{(j)}$ for each child $j$
Let $S^{(k)}$ be the smallest sample value for the stretch, and $S^{(l)}$ the largest one
while $k \neq l$ do
    $mid = \lceil (l+k)/2 \rceil$
    if $\sum_j n^{(j)}_{\text{new}}\left(S^{(mid)}\right) \geq N_{\text{new}}$ then
        $k = mid$
    else
        $l = mid - 1$
return $S^{(k)}$
```

The new application can thus be scheduled locally in the subtree, but all applications in this subtree will see their stretch increase to $S^{(i)}$, which might be large. In some cases, this would lead to an unacceptable load imbalance in the platform. To prevent such situations, another information is used: the minimum stretch. The minimum stretch of the platform is periodically aggregated from the leaf nodes to the root node, and then spread from the root to all other nodes. Since we simply compute and transfer a minimum value, its size is negligible, and this information may be included in any other update messages.

Once the minimum local stretch $S^{(i)}$ is computed, we accept this local solution if and only if $S^{(i)} \leq B \times S_{\text{min}}$, for a given bound B. Otherwise, the entire request is sent to the father to look for a better allocation. This implements a tradeoff between performance and load-balancing, so that small applications will remain local, and large applications can go up the tree until they induce an acceptable slowdown. If parameter B is close to one, most requests would need to go up the tree until enough execution nodes are at sight; if it is larger, the ratio between the maximum and the minimum stretch in the whole platform may be large, and the allocation less fair.

Once a routing node finds a suitable value for the minimum stretch on its local subtree, the tasks in the request are ready to be split among the children. This is done following the values of $n(S)$ in the availability summary of each child: child j gets a share of the total number of tasks proportional to its availability $n^{(j)}(S)$. A new request is then sent to each child, with the characteristics of the application, and the number of tasks it is in charge of. This request is treated as previously, except that it cannot be rejected and forwarded to its father.

When a routing node fails, the information on the availability of its subtree is lost. As we pointed out in section 3.2, we assume that our scheduler is based on a tree-based network overlay which already guarantees a high probability of reaching a stable state when a node fails, by reconstructing and balancing the tree. Usually, the tree overlay will recover by replacing the failed node by another one and performing some balancing operations, like in [12]. While some routing nodes may change their positions, the sets of execution nodes that lay under their branches are maintained. Thus, the availability summaries must be recalculated only in such nodes, and possibly in the ancestor nodes up to the root. The cost of this process is not higher than the cost of a normal update, so the impact of such a the failure on the global scheduling is mostly limited by the time needed to recover the overlay. During the update of availability summaries, some applications might be scheduled using improper information, however this effect is mitigated if we rely on an overlay which only performs local moves to balance the tree.

7 Experimental evaluation

Our proposal has been implemented in a platform simulator in order to test and validate its performance. Using simulations allows us to conduct reproducible experiments, so as to compare performance of several schedulers on the same scenarios. Our simulations do not only provide performance measurements of the decentralized scheduler, but also several results on the impact of its algorithms and protocols on the network and computational resources of the platform. As we show, these results demonstrate the benefits of adopting a decentralized scheduler in a desktop grid platform over its centralized counterpart.

7.1 Simulation settings

Simulations have been performed on networks including between 50 and 1000 nodes, where execution nodes have a computing power in the interval [1000, 3000] with steps of 200, in millions of instructions per second. The network is fully connected, i.e., there is a link between any two nodes, with a mean link delay of 50 milliseconds and link bandwidth of 1 megabit per second. This scenario represents a desktop grid of home computers with a modest Internet connection, with different sizes. Workload is generated as a Poisson process, with a mean inter-arrival time of 10 seconds. With that time, applications arrive while others are still being executed. The number of tasks per application is ten times the size of the network, with a random variation of $\pm 20\%$.

Our decentralized scheduler is tested with and without failures. They occur in each node as a Poisson process of rate one failure each four hours. Failures are instantaneous, so that nodes recover immediatelly, but with reset state. A failed node retains no availability summaries from its children, and all the tasks that were waiting in its queue are aborted. The tree is supposed to recover automatically.

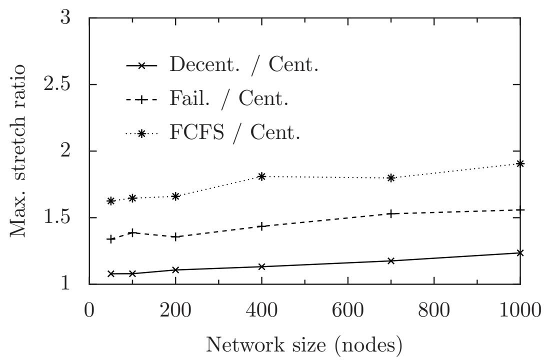
Along with the simulation of our decentralized scheduler, two centralized schedulers have also been tested with the same set of applications and under the same conditions. Both simulate an online centralized scheduler with perfect information about the execution nodes. The first one tries to minimize the maximum stretch, as in the decentralized version; we call it MinCent for short. The second one implements a typical FCFS centralized scheduler, similar to the one used by other popular desktop grid platforms, and thus is not expected to achieve very good fairness among applications. The comparison of our decentralized scheduler against both centralized models is interesting because it shows the inevitable performance loss by the use of decentralized information compared to the global scheduling algorithms, and the performance gain compared to classical schedulers not focusing fairness.

7.2 Simulation results

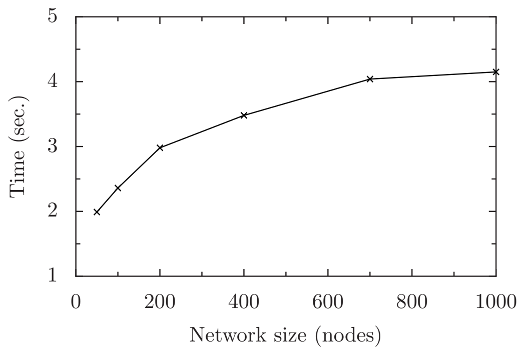
In order to compare the performance of the different schedulers, we issued several simulations for various network sizes, with one hundred applications each, registering the maximum stretch and calculating the mean value for each network size. Figure 4(a) plots this values for the decentralized and FCFS schedulers, relative to the values obtained with MinCent, against network size. As it can be seen, with one thousand nodes, the performance lost in the decentralized scheduler without failures is still under 25%. As the number of nodes increases, the performance is slightly reduced, because in a higher tree, the information used by the upper levels is less detailed. In the scenario with failures, the performance is noticiably reduced, as expected, but still better than the one achieved by the FCFS centralized scheduler. As expected, the classic scheduling policy behaves much worse in terms of fairness, providing a maximum stretch almost twice higher under the same conditions.

On the other hand, the appearance of failures has the side effect of reducing the difference between the stretch of different applications, actually providing better fairness. For instance, for a network of one thousand nodes, the stretch of the applications without failures was in the interval $[220 \cdot 10^{-8}, 10876 \cdot 10^{-8}]$, while in the case of failures it was in the interval $[10303 \cdot 10^{-8}, 12533 \cdot 10^{-8}]$. We deduce that this is due to the fact that the scheduler is able to further adjust the stretch of the applications with the resubmitted tasks.

The good performance ratio against the centralized scheduler with minimum stretch objective shown in the previous results is due partly to the dynamic and fast update of availability information throughout the tree when tasks arrive or finish at the execution nodes. Having up-to-date availability information is decisive for the global scheduler efficiency. Figure 4(b) shows the maximum update time needed for different network sizes, with an update rate of 10000 bytes per second. As it can be seen, for the network of one thousand nodes, a change in the local scheduler of any execution node can be propagated



(a) Maximum stretch obtained by the decentralized and FCFS centralized schedulers, relative to MinCent, against network size.



(b) Maximum update time needed in networks of different sizes, for an update rate limit of 10000 Bps.

Figure 4: Experimental results.

Update rate (Bps)	2500	5000	10000	20000	40000
Max. update time	12.3	6.61	3.68	2.33	1.81
Mean link usage	2.41%	3.78%	4.28%	5%	5.84%
Peak link usage	11.41%	16.43%	24.88%	40.45%	72.63%

Table 1: Update time and link bandwidth usage for different update rate limits in a network of 1000 nodes.

in less than four seconds. As expected, higher network sizes make the distribution of the update time shift to higher values. Even though, the difference is very little due to the logarithmic increase of the tree height.

With higher update rate limits, shorter times can be achieved, nearly in inverse proportion. However, the update rate limit is a critical parameter since we have recorded in the simulation test that update messages represent more than 95% of the traffic, due to the size of an availability summary, so it must be increased carefully when better reactivity is needed. Table 1 presents update time, mean and peak link bandwidth usage for different update rate limits in a network of 1000 nodes. While mean link usage is quite low in every case, peak usage rapidly increases. From a general point of view, mean usage shows that the traffic generated by the platform protocols to schedule the submitted applications is very low, even at the root node of the tree. However, traffic peaks may be to intrusive for a desktop grid. Since the corresponding gain in update time is small, we consider the value of 10000 bytes per second adequate for the kind of links used in the simulation.

The asymmetry of the tree causes higher levels to cope with higher peak bandwidth usage than lower ones. However, the increase at each level is not constant, since after a certain level the update rate limit avoids peak usage to continue growing. For instance, with an update rate limit of 10000 bytes per second in a network of 1000 nodes, the peak bandwidth usage was between 24.5% and 25% in any level over the third lowest one.

Moreover, although not used in the scheduling algorithm, the traffic generated by task data transfers has been measured and compared to the traffic generated by the platform protocols. We use a task data size of 512 kilobytes, low enough to represent an application which transmission time does not affect scheduling, and we assume that repositories where data is stored are not limited in bandwidth. Under these conditions, on each node, the average data traffic is still 10 times larger than the traffic generated by the platform protocols.

8 Conclusions and future work

In this paper, we have focused on the problem of scheduling concurrent bag-of-tasks applications on desktop grids. We have proposed a decentralized scheduling algorithm, which makes it particularly convenient for large-scale distributed environments. The objective of our scheduler is to ensure fairness among applications, by minimizing the maximum slowdown, or stretch, of all applications.

Through extensive simulation tests, we have compared this scheduler to an online centralized version that also minimizes the maximum stretch, but need a perfect knowledge of the platform, and to a classical FCFS scheduler, which is commonly used on desktop grids. We prove that the performance loss compared to the centralized scheduler is reasonable (below 25%), and that the achieved fairness is much better than with FCFS, even under frequent node failure. We also carefully studied the resource usage of our algorithm, which proves to use a significantly small quantity of network resources at each node of the platform. Moreover, the CPU consumption at each node of the platform of our approach is very low, due to the simplicity of the proposed algorithms. Thus, our decentralized algorithms has a very low overhead together with a great flexibility and robustness, which makes it very suited for desktop grid platforms.

Our future works include simulations to study the adaptation of our scheduler to communication-intensive applications, by taking file size into account when allocating tasks onto the platform. We also intend to improve our scheduler with more complex overlays proposed for peer-to-peer platforms.

Acknowledgment

Javier Celaya and his work has been supported by the CICYT DPI2006-15390 project of the Spanish Government, grant B018/2007 of the Aragonese Government, grant TME2008-01125 of the Spanish Government, and the GISED, group of excellence recognized by the Aragonese Government.

References

[1] Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-Grid: A Self-organizing Structured P2P System. SIGMOD Rec. 32(3), 29–33 (2003)

[2] Al-Azzoni, I., Down, D.G.: Dynamic scheduling for heterogeneous Desktop Grids. In: GRID '08: Proceedings of the 9th IEEE/ACM International Workshop on Grid Computing. pp. 136–143 (2008)

[3] Al-Kiswany, S., Ripeanu, M., Vazhkudai, S.S., Gharaibeh, A.: stdchk: A Checkpoint Storage System for Desktop Grid Computing. In: ICDCS '08: Proceedings of the 28th International Conference on Distributed Computing Systems. pp. 613–624. IEEE Computer Society, Washington, DC, USA (2008)

[4] Anderson, D.P.: BOINC: A System for Public-Resource Computing and Storage. In: GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing. pp. 4–10. IEEE Computer Society, Washington, DC, USA (2004)

[5] Anglano, C., Brevik, J., Canonico, M., Nurmi, D., Wolski, R.: Fault-aware scheduling for bag-of-tasks applications on desktop grids. In: GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing. pp. 56–63. IEEE Computer Society, Washington, DC, USA (2006)

[6] Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Marchal, L., Robert, Y.: Centralized versus Distributed Schedulers for Bag-of-Tasks Applications. IEEE Trans. Parallel Distrib. Syst. 19(5), 698–709 (2008)

[7] Benoit, A., Marchal, L., Pineau, J.F., Robert, Y., Vivien, F.: Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. to appear in IEEE Transactions of Computers (2009), PrePrint available online at `http://doi.ieeecomputersociety.org/10.1109/TC.2009.117`

[8] Brasileiro, F., Araujo, E., Voorsluys, W., Oliveira, M., Figueiredo, F.: Bridging the High Performance Computing Gap: the OurGrid Experience. In: CCGRID '07: Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid. pp. 817–822. IEEE Computer Society, Washington, DC, USA (2007)

[9] Celaya, J., Arronategui, U.: YA: Fast and Scalable Discovery of Idle CPUs in a P2P network. In: GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing. pp. 49–55. IEEE Computer Society, Washington, DC, USA (2006)

[10] Celaya, J., Marchal, L.: A fair distributed scheduler for bag-of-tasks applications on desktop grids. Resarch Report RRLIP2010-07, LIP (2010), available at `http://graal.ens-lyon.fr/~lmarchal`

[11] Choi, S., Kim, H., Byun, E., Baik, M., Kim, S., Park, C., Hwang, C.: Characterizing and Classifying Desktop Grid. In: CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid. pp. 743–748. IEEE Computer Society, Washington, DC, USA (2007)

[12] Jagadish, H., Ooi, B.C., Vu, Q.H., Zhang, R., Zhou, A.: VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes. In: Proceedings of the 22nd International Conference on Data Engineering, 2006. ICDE '06. p. 34 (2006)

[13] Kondo, D., Chien, A.A., Casanova, H.: Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. p. 17. IEEE Computer Society, Washington, DC, USA (2004)

[14] Legrand, A., Su, A., Vivien, F.: Minimizing the stretch when scheduling flows of biological requests. In: SPAA '06: Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures. pp. 103–112. ACM, New York, NY, USA (2006)

[15] Li, M., chien Lee, W., Sivasubramaniam, A.: DPTree: A Balanced Tree Based Indexing Framework for Peer-to-Peer Systems. In: ICNP '06: Proceedings of the Proceedings of the 2006 IEEE International Conference on Network Protocols. pp. 12–21. IEEE Computer Society (2006)

[16] Litzkow, M.J., Livny, M., Mutka, M.W.: Condor-a hunter of idle workstations. In: ICDCS: Proceedings of the 8th International Conference on Distributed Computing Systems (1988)

[17] Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: Adopt: asynchronous distributed constraint optimization with quality guarantees. Artif. Intell. 161(1-2), 149–180 (2005)

[18] Ramamritham, K., Stankovic, J.A., Zhao, W.: Distributed Scheduling of Tasks with Deadlines and Resource Requirements. IEEE Trans. Comput. 38(8), 1110–1123 (1989)

[19] seti@home, `http://setiathome.berkeley.edu/`

[20] Shirts, M., Pande, V.: Screen Savers of the World Unite! Science 290(5498), 1903–1904 (2000)

[21] da Silva, D.P., Cirne, W., Brasileiro, F.V.: Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In: Proceedings of the 9th International Euro-Par Conference. pp. 169–180 (2003)

[22] Viswanathan, S., Veeravalli, B., Robertazzi, T.G.: Resource-Aware Distributed Scheduling Strategies for Large-Scale Computational Cluster/Grid Systems. IEEE Trans. Parallel Distrib. Syst. 18(10), 1450–1461 (2007)

[23] World community grid, `http://www.worldcommunitygrid.org/`

[24] Zhou, D., Lo, V.M.: WaveGrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system. In: IPDPS '06: Proceedings of the 20th International Parallel and Distributed Processing Symposium (2006)