

A Novel Resource-Driven Job Allocation Scheme for Desktop Grid Environments*

Paolo Bertasi, Alberto Pettarin, Michele Squizzato, and Francesco Silvestri

Department of Information Engineering,
University of Padova, Padova, ITALY
{bertasi,pettarin,scquizza,silvest1}@dei.unipd.it

Abstract In this paper we propose a novel framework for the dynamic allocation of jobs in grid-like environments, in which such jobs are dispatched to the machines of the grid by a centralized scheduler. We apply a new, full resource-driven approach to the scheduling task: jobs are allocated and (possibly) relocated on the basis of the matching between their resource requirements and the characteristics of the machines in the grid. We provide experimental evidence that our approach effectively exploits the computational resources at hand, successfully keeping the completion time of the jobs low, even without having knowledge of the actual running times of the jobs.

1 Introduction

Groups of distributed, heterogeneous computational resources, called *Grids* [9], have recently emerged as popular platforms to tackle large-scale computationally-intensive problems in science, engineering, and commerce. The desktop grid computing technology permits to exploit the idle computational resources of a large amount of non-dedicated heterogeneous machines within a single organization or scattered across several administrative domains.

In order to properly exploit the potential of these grid systems, key services such as resource management and scheduling are needed. Indeed, effectively matching tasks with the available resources is a major challenge for a grid computing system because of the heterogeneous, dynamic and autonomous nature of the grid, and a great deal of research concerning scheduling strategies capable of fully exploiting computational grids has been conducted.

In this paper, we propose a novel allocation scheme in which job and resource characteristics are captured together in the scheduling strategy, without resorting to the knowledge of the running times of the jobs, which usually are not known in advance. To this end, each machine that joins the grid is represented

* This work was supported, in part, by the European Union under the FP6-IST/IP Project AEOLUS, and by University of Padova under the Strategic Project STPD08JA32. Part of this work was done while the second author was visiting the Department of Computer Science of Brown University, USA, supported by “Fondazione Ing. Aldo Gini”, Padova, Italy.

by a d -dimensional *speed vector*, whose components are numerical values which quantify features of the system such as CPU clock, and disk/network bandwidth. Similarly, each user submitting a job characterizes the computational properties of his task by giving the estimated percentages of how the operations of the job will “distribute” among the features of the machines. For example, and with respect to the three aforementioned features, a pure CPU intensive task might be described with a triplet similar to $\langle 1, 0, 0 \rangle$, while a job dealing with a local large data set might be represented by $\langle 0.4, 0.6, 0 \rangle$. Then, a quantity similar, in spirit, to the inner product between the vector of the job and the speed vector of a machine m provides a quantitative measure of the suitability of the machine for the job, with the job being dispatched to the machine which guarantees the highest score. Allocation is *centralized*, that is, one node in the system acts as a scheduler and makes all the load balancing decisions, and *on-line*, as jobs must be assigned upon their arrival. Moreover, scheduling is *dynamic*, since we allow the grid scheduler to migrate a job (we assume we are dealing with *preemptable* jobs) as soon as the completion of a job in a host or a variation of the load due to the machine owner might make some re-assignments fruitful. (Hence, load re-balancing is event-driven, while in most systems it is simply performed through periodic rescheduling.) We consider a finite-size temporal window of job arrivals with the implicit goal of keeping as low as possible the *makespan* of the schedule, that is, the completion time of the job that finishes last.

Previous work already showed how taking into account all features of the jobs in the scheduling activity leads to good performances. However, we argue that no one of the existing resource-aware allocators relies upon a score mechanism for machine-job pairs which is effective (in that it fully leverages knowledge about job and machine characteristics to assign the former to the host that best meets the user requirements), fair, dynamic, and easy to use at the same time.

Related work. Since most variants of the task scheduling problem are NP-complete [10], a great deal of effort has been devoted to the development of approximation and heuristic algorithms (see, e.g., [12,17,20,5,15]). However, these works make the strong assumption that perfect knowledge of how long each job will run is known at the time of scheduling, while our strategy does not require this knowledge; indeed users’ runtime estimates are notoriously inaccurate [7], and it seems that users are generally incapable of providing more accurate estimates [14], with the problem being worsened by the heterogeneity of the machines of a grid. Moreover, the estimates required by our allocator to the user are completely machine-independent.

For these reasons, most real grid brokering strategies rely on a suitable mapping of user jobs to hosts according to the requirements of the former and to the properties of the latter. A number of grid middleware and management mechanisms have been designed to this end. Condor [18] provides a general resource selection mechanism based on the ClassAd [21], a language that allows resource owners to describe their resource and users to describe resource requests for their jobs. Specifically, all machines in the Condor pool use a resource offer ad to advertise their resource properties, both static and dynamic, such as CPU

type, CPU speed, available RAM memory, physical location, and current average load, and users specify a resource request ad when submitting a job. The request consists of the set of minimal resources needed to run the job, along with a field in which the user specifies the function to be maximized by the broker. (ClassAd has also been extended to allow users to specify aggregate resource properties, e.g., in [19].) Condor acts as a broker by matching user requests with appropriate resources. However, we notice that our approach is simpler to use for the user, as the information required for the matching is simply an estimated repartition of the machine capabilities to be exploited. In Globus [8], users describe required resources through a resource specification language (RSL) that is based on a predefined schema of the resources database. The task of mapping specifications to actual resources is performed by a resource allocator, which is responsible for coordinating the allocation and management of resources at multiple sites. The RSL allows users to provide very sophisticated resource requirements (while no analogous mechanism for resources exists), but this comes at the price of ease-of-use. The Application Level Scheduling project (AppLeS [3]) uses the performance model provided by users to schedule applications. Key to the AppLeS approach is that resources in the system are evaluated in terms of predicted capacities at execution time, as well as their potential for satisfying application resource requirements. In the Nimrod/G system [1] the scheduling policy is driven by an economic model which supports user-defined deadline and “budget” constraints for schedule optimizations, and maps a job to the lowest-cost resource able to meet its deadline. Again, an effective utilization by the user is not immediate. In the work of Khoo *et al.* [13], jobs and resources are mapped in a multi-dimensional space, and nearest neighbor searches are conducted by the scheduling algorithm, with a job being dispatched to its nearest machine in such a space. While being quite similar to ours, their strategy does not consider job relocation.

Our contribution. In this paper we introduce a new resource-driven allocation scheme for grid environments, in which the scheduling mechanism assigns jobs to machines that are best suited for their resource requirements without knowing their actual running times. We describe two different schemes: the first, called GREEDY allocation scheme, greedily maps and relocates job to the machine which represents the best match, while the second, termed SOCIAL allocation scheme, performs the choice that best affect the “social welfare”. We set up a grid simulation environment to demonstrate the efficacy of the proposed scheduling solution. Indeed, experimental results give evidence that our algorithms perform effectively the allocation task, that is, the allocation is fair, balanced, and the resulting makespan is kept low. Moreover, we show that the second outperforms the first in many cases of interest for real-life scenarios.

Paper organization. The rest of the paper is organized as follows. Section 2 describes the model and our algorithms for allocation and relocation. In Section 3, experimental results are presented which provide evidence of the effectiveness of our approach. Finally, in Section 4 we draw some conclusions and discuss directions for future work.

2 The Framework

In this section we first provide a simple but effective model of a desktop grid environment, and then describe two procedures for job allocation and relocation which assign a job to the machine that best suits the job requirements according to two different criteria: a selfish one (from the point of view of the job), which we simply call GREEDY allocator, and a “more altruistic” one, the SOCIAL allocator.

2.1 The Model

We represent a computational grid as a collection of heterogeneous machines: each machine $m \in \mathcal{M}$ can perform $d > 0$ types of “real-world” operations (e.g., CPU instructions, read/write data from/to disk, receive/send data through the network) at rates defined by its *speed vector* \mathbf{S}_m . Specifically, \mathbf{S}_m is a d -dimensional vector where component $\mathbf{S}_m[i]$, for $0 \leq i < d$, represents the number of *type- i* operations that can be performed by m in a time unit (e.g., CPU frequency, disk, network bandwidth). Each machine performs at most one operation at a given time instant, that is, we ignore any form of concurrency among operations: this is a worst-case scenario since in general some operation types can be (partially) performed in parallel on modern machines. To model the fact that some of the computational power of machine m is used by its *owner*, we introduce the *owner load* λ_m , with $0 \leq \lambda_m \leq 1$, which represents the fraction of resources of m devoted to the owner’s needs: in other words, we suppose that all the rates in \mathbf{S}_m are multiplied by a factor $(1 - \lambda_m)$. We allow the owner load to change dynamically over time. Note that we are implicitly assuming that the owner load impacts on *each* component of the machine. Of course, this might not be true in specific scenarios, for example when the machine owner always requires only a given type of resources (e.g. she just needs the CPU but not the disk resources). However, since the grid does not know the owner’s computational requests, we choose to simply scale down *all* the components of a machine by the same factor. It deserves to be remarked that in most of previous work, a machine is either completely available (i.e., $\lambda_m = 0$) or completely not available (i.e., $\lambda_m = 1$), and any intermediate status is not taken into account.

A job j , which consists of $\ell_j \geq 1$ operations of the various types, is described by its *composition vector*, a d -dimensional unit vector whose component $\mathbf{C}_j[i]$, for $0 \leq i < d$, represents the percentage of type- j operations, measured as multiples of $\mathbf{W}[i]$, where \mathbf{W} is the d -dimensional *weight vector*. The weight vector, which we assume to fix a priori, can be seen as a sort of “operation-exchange” unit system between the various components. Indeed, it implicitly defines a common “logical concurrency” between the various components, where one “logical operation” corresponds to $\mathbf{W}[i]$ “real-world” operations of the i -th component. Then, job j contains at most $\ell_j \cdot \mathbf{W}[i] \cdot \mathbf{C}_j[i]$ type- i operations. We observe that the model can be defined without the vector \mathbf{W} , however weights are needed for tuning the composition vector to reflect the actual effect on performance of each component. For example, suppose type-0 operations are numerous but fast and type-1 operations are few but slow: if \mathbf{W} is not used (i.e., $\mathbf{W}[i] = 1$ for each i),

we have $\mathbf{C}_j[0] \gg \mathbf{C}_j[1]$ even if their influences on performance are comparable. Intuitively, \mathbf{C}_j characterizes the computational properties of job j by giving the estimated percentages of utilization of each machine subsystem.

A machine can execute an arbitrary number of jobs¹, which are performed according to a round robin scheduler which assigns fixed-size time slices to each job, handling all processes without priority. For simplicity, we assume the time slice to be small in comparison to the overall task length. Under these assumptions, the *execution time* $t(j, m)$ of a new job j which starts on machine m can be reasonably estimated by

$$t(j, m) = \frac{\ell_j(n_m + 1)}{1 - \lambda_m} \sum_{i=0}^{d-1} \frac{\mathbf{W}[i]\mathbf{C}_j[i]}{\mathbf{S}_m[i]}, \quad (1)$$

where n_m is the number of jobs other than j running on m . (Note that both n_m and λ_m change dynamically, but we omit their dependence on time for ease of notation.) Clearly, the execution time of a job j has to be proportional to its length ℓ_j and it has to grow at the same rate of $(n_m + 1)$, due to the fair resource sharing mechanism. Conversely, it must be inversely proportional to the fraction $(1 - \lambda_m)$ of the machine power not utilized by the owner and thus at the grid user's disposal. Finally, the summation is justified by the assumption that the execution time is split among the various components without overlapping.

Whenever n_m or λ_m change during the execution of j , we first calculate the number of remaining operations ℓ'_j , and then update the estimated execution time by replacing ℓ_j with ℓ'_j in Equation (1). (We assume that the composition of the non-executed operations reflects the composition vector \mathbf{C}_j .) Throughout the paper we denote by \mathcal{J}_m the set of jobs running on machine m at the time instant under consideration.

When a new job is submitted to the grid, it is handled by the *allocator*, which reads its composition vector and assigns the job to a suitable machine according to the allocation scheme of choice². The allocation is *dynamic* because we allow relocation of jobs upon the occurrence of events that modify the load of a machine, in particular when a machine completes the execution of a job, or when a owner load varies. In both cases, we suppose that the involved machine notifies the allocator of the change taking place.

It is important to recall that the parameters that characterize machines and jobs can be quickly estimated in a real scenario. The speed vector \mathbf{S}_m of a new machine m can be determined automatically, reducing the burden of its owner willing to share the machine, through a *microbenchmarking* suite such as that of [4]: specifically, once a machine m joins the grid, the system performs a round

¹ Clearly, a job j cannot be executed on a machine m whose owner load λ_m is 1 (i.e., the machine is not available). Furthermore, if $\mathbf{S}_m[i] = 0$ for some $0 \leq i < d$, then machine m cannot execute a job j with $\mathbf{C}[i] \neq 0$; for this reason, we suppose that $0/0 = 0$ (e.g., in the subsequent equation).

² For simplicity, in our model the allocation task is performed in a centralized fashion, however nothing impedes to implement it in a distributed way, for example, to improve the robustness of the whole system.

of microbenchmarking to derive the peak performance of m , which will be used to derive its \mathbf{S}_m . The same microbenchmarking suite, or faster heuristics on the CPU usage, can be used periodically for computing the owner load λ_m of the machine. To further reduce the specification burden to the user, the composition vector \mathbf{C}_j of a new job j may be chosen by the user submitting the job by associating it to a label, corresponding to a certain composition vector. The label might be selected from a small, predefined set of labels, each related to the most common job types (e.g., CPU intensive jobs, jobs dealing with local large data sets, etc.), thus relieving the user of explicitly specifying the composition vector for his job (which additionally requires the knowledge of the weight vector). The aforementioned set of labels and the weight vector \mathbf{W} can be determined in the initial set-up of the grid environment. We argue that we do not require the user submitting a job j to provide an estimation of ℓ_j , as the allocators we are going to describe do not rely upon its knowledge.

2.2 Allocation Procedures

In this section we describe two allocation procedures for our model which differ on the score function used to assign jobs to machines. When a new job j arrives, both allocators assign j to the machine m maximizing a given score function $f(j, m)$, which is differently defined in the two procedures. A key element in our allocators is the notion of affinity which is a measure of the suitability of a machine to execute a certain job. The *affinity* $\tau(j, m)$ of job j on machine m is defined as

$$\tau(j, m) = \frac{1 - \lambda_m}{(n_m + 1) \sum_{i=0}^{d-1} \mathbf{W}[i] \mathbf{C}_j[i] / \mathbf{S}_m[i]},$$

where n_m is the number of other jobs that are executing on m . The affinity depends on the time instant in which it is computed, since n_m and λ_m change dynamically; however, for notational simplicity, we omit the dependence on time from $\tau(j, m)$. As one can easily recognize, the affinity and the estimated completion time are related by the following formula:

$$t(j, m) = \frac{\ell_j}{\tau(j, m)}. \quad (2)$$

The GREEDY allocator relies on Equation (2), and simply sets its score function to $f(j, m) = \tau(j, m)$. Therefore, job j is assigned to the machine maximizing its affinity, and thus minimizing its execution time (despite of the allocator being unaware of the actual job length ℓ_j). However, this selfish approach ignores the fact that the execution times of the jobs running on m grow (or, by our definition, their affinities decrease).

In order to reduce the latter negative effect, we can correct the score function as follows, obtaining what we dubbed SOCIAL allocator:

$$f(j, m) = \tau(j, m) - \sum_{k \in \mathcal{J}_m} \frac{\tau(k, m)}{n_m + 1}.$$

The term $\tau(k, m)/(n_m + 1)$ denotes the decrease in affinity of job $k \in \mathcal{J}_m$ (i.e., already executing on m) if the new job j is assigned to m . The above equation provides a trade-off between the selfish approach where the job minimizes its execution time, and a social approach where the job is assigned to the machine where the execution times of preexisting jobs do not increase excessively.

In Section 3 we analyze experimentally the two approaches without relocation, and provide evidence that the makespan obtained with the SOCIAL allocator is in general better than the one with the GREEDY one.

2.3 Relocation Procedures

In this section we describe two relocater procedures, namely the GREEDY and SOCIAL relocators, which are similar to their allocation counterparts. They act similarly when a machine status changes, but differ on the implementation of function $f(j, m, m')$, which is used as a score for evaluating the migration of job j from machine m to machine m' . We describe how $f(j, m, m')$ is implemented by the two procedures after explaining the relocation mechanism.

The events that cause the invocation of the relocater are the following:

- The owner load of machine m increases. In this case, the relocater migrates a job in \mathcal{J}_m into another machine m' in order to reduce the effect of the variation; job $j \in \mathcal{J}_m$ and machine m' are chosen so that $f(j, m, m')$ is maximized.
- The owner load of machine m decreases or a job in m terminates its execution. In this case, the relocater moves a job j from machine m' into m in order to use the available computational resources of machine m and at the same time to reduce the load of machine m' . Machine m' and job $j \in \mathcal{J}_{m'}$ are chosen so that function $f(j, m', m)$ is maximized.

To ensure that a prospective action leads to an actual improvement of the system state, function $f(j, m, m')$ is expressed as a relative gain with respect to the previous system state. This gain has to be greater than a given constant threshold $\theta > 0$. The meaning of θ is easy to understand: the lower θ , the more likely relocations occur, and vice versa. This stipulated relative threshold aims at modeling the cost of job migration, which also includes the intrinsic overhead of each preemption-and-resume step. Moreover, after a single event, the above procedure can be iterated until no improvement can be obtained or the maximum number of iterations N is reached (being N an a priori fixed constant).

In the GREEDY relocater, $f(j, m, m')$ is given by

$$f(j, m, m') = \frac{\tau(j, m') - \tau(j, m)}{\tau(j, m)},$$

where the affinities are computed at the instant where the load of m changes. In other words, the GREEDY relocater moves the job j which maximizes its relative affinity increment, that is, the job reaches the biggest relative decrease in execution time. As for the GREEDY allocator, this procedure does not take

into account jobs already present on the machine where the job is migrated, whose execution times increase (and affinities decrease).

On the contrary, the SOCIAL relocater takes into account also the difference between the increase in affinity of jobs on m (since the number of jobs decreases) and the decrease in affinity of jobs on m' (since the number of jobs increases). The proposed score function takes the form

$$f(j, m, m') = \frac{\alpha_0(\Sigma_m - \Sigma'_m) + \alpha_1(\Sigma_{m'} - \Sigma'_{m'})}{\Sigma_m + \Sigma'_m},$$

where Σ'_m (respectively, $\Sigma'_{m'}$) and Σ_m (respectively, $\Sigma_{m'}$) denote the sum of affinities of jobs in m (respectively, m') before and after the migration of job j from m to m' . The coefficients

$$\alpha_0 = \frac{|\mathcal{J}_m|}{|\mathcal{J}_m| + |\mathcal{J}_{m'}|} \quad \text{and} \quad \alpha_1 = \frac{|\mathcal{J}_{m'}|}{|\mathcal{J}_m| + |\mathcal{J}_{m'}|}$$

are used for balancing the number of jobs among machines when m and m' contain similar workloads.

In the next section we show that the GREEDY and the SOCIAL relocaters attain the same performances, independently of the adopted allocation procedure.

3 Experimental Results

In this section we experimentally compare the allocators and relocaters described in previous section with a simple allocation scheme, referred to as MIN-NUM. The MIN-NUM allocator assigns a new job to the machine with the minimum number of running jobs at the arrival time, independently of job and machine characteristics. Similarly, the MIN-NUM relocater invoked on machine m moves a job from the machine m' with maximum $\mathcal{J}_{m'}$ to m if the load on m decreases (i.e., a job terminates its execution, or the owner load decreases), or moves a job from machine m into the machine m' with minimum \mathcal{J}_m if the load on m increases (i.e., the owner load increases); this invocation is executed at most N times for each invocation, where N is a suitable constant. This allocator scheme can be efficiently implemented, however it performs poorly as shown in the following examples. All the experiments are carried out through a Java simulator, whose source code might be obtained upon request to the authors.

We consider three types of operations: the components of a speed vector represent, in order, CPU frequency (in GHz), disk bandwidth (in MB/s), and network bandwidth (in KB/s). We consider two machine sets. The first one, named *synthetic grid*, consists of four machines characterized by the speed vectors $\mathbf{S}_0 = \langle 4, 100, 250 \rangle$, $\mathbf{S}_1 = \mathbf{S}_2 = \langle 2, 100, 250 \rangle$, and $\mathbf{S}_3 = \langle 1, 800, 250 \rangle$. The second one, named *AEOLUS grid*, models the AEOLUS testbed [2] and consists of 70 machines, whose speed vectors are given in Table 3 in the Appendix. The synthetic grid is used to enlighten some properties of allocators and relocaters,

while the simulations of the AEOLUS grid provide evidence of their performance in a real-world scenario.

To the best of our knowledge, publicly available workloads like those in Feitelson’s Parallel Workloads Archive [6] do not consider job features such as those required by our framework, and hence the composition vectors used in our experiments are artificial, and described in Table 1. We note that \mathbf{C}_0 denotes a CPU intensive job, \mathbf{C}_1 a generic job which uses all operations, \mathbf{C}_2 a network intensive job, \mathbf{C}_3 and \mathbf{C}_4 disk intensive jobs. For simplicity, we say that a job is of type i , for $0 \leq i \leq 4$, if its composition vector is \mathbf{C}_i . The weight vector used in the experiments is $\mathbf{W} = \langle 2 \cdot 10^{-5}, 10^{-1}, 10^{-1} \rangle$. Since some studies (e.g., [16,11]) show that durations of real jobs are distributed according to a power law, we generate job lengths using a discrete representation of a power law.

Type	CPU	Disk	Network	Description
\mathbf{C}_0	1.0	0.0	0.0	CPU intensive
\mathbf{C}_1	0.7	0.1	0.2	Generic (all operations)
\mathbf{C}_2	0.6	0.0	0.4	Network intensive
\mathbf{C}_3	0.5	0.5	0.0	Disk intensive I
\mathbf{C}_4	0.2	0.8	0.0	Disk intensive II

Table 1: Job composition vectors adopted in this section.

We remind that, when the load of a machine changes, the GREEDY and SOCIAL relocators perform job migration until the relative gain of the score function $f(j, m, m')$ is bigger than θ (e.g., 5%), and no more than N job relocations might occur. In Figure 1, we analyze the behavior of GREEDY and SOCIAL relocators for different values of N and θ , and of MIN-NUM for different N ’s. Each relocator is associated with its respective allocator. We use the synthetic grid described above, with the owner loads set to 0, and jobs described by composition vectors \mathbf{C}_0 and \mathbf{C}_1 , which arrive uniformly in the time interval $[0, 100]$ s and whose lengths are generated according to a three-step discretization of a power law distribution (the exact description³ is available in section “Job set 0” of Table 2). We notice that all the relocators exhibit small fluctuations (about 1%) when N or θ changes, and the GREEDY and SOCIAL relocators are almost equivalent. For these reasons in the following experiments we set $N = 3$ for decreasing the computational cost of relocation, and $\theta = 10\%$ for justifying the migration cost (moving jobs with small score increments is not convenient since the migration costs may be bigger than the execution time saved after relocation). The difference between our relocators and the MIN-NUM one is small in the analyzed data set, however we later show that in a more general scenario the gap considerably increases. We performed other experiments, which are not reported for lack of space, where we analyze any allocator/relocator combination: in all cases the makespan remains almost constant changing N and θ and the GREEDY and SOCIAL relocators provide the best makespans independently of the used allocator.

³ In the paper we denote by $\mathcal{N}(\mu, \sigma)$ a Gaussian random variable with mean μ and standard deviation σ , and by $\mathcal{U}(i, j)$ an uniform random variable in the interval $[i, j]$.

Job type	Number of jobs	Length	Arrival time (in s)
Job set 0 (Figure 1)			
C_0	200	24% with length $\mathcal{N}(400k, 8k)$, 38% with length $\mathcal{N}(200k, 4k)$, 38% with length $\mathcal{N}(100k, 2k)$	$\mathcal{U}(0, 100)$
C_1	800	24% with length $\mathcal{N}(100k, 4k)$, 38% with length $\mathcal{N}(50k, 2k)$, 38% with length $\mathcal{N}(25k, 1k)$	$\mathcal{U}(0, 100)$
Job set 1 (Figure 2)			
C_0, C_4	500 per type	20% with length $\mathcal{N}(500k, 40k)$, 38% with length $\mathcal{N}(250k, 20k)$, 38% with length $\mathcal{N}(125k, 10k)$	$\mathcal{U}(0, 1000)$
Job set 2 (Figures 3 and 4)			
C_0, C_1 C_2, C_3	500 per type	20% with length $\mathcal{N}(250k, 75k)$, 30% with length $\mathcal{N}(100k, 30k)$, 30% with length $\mathcal{N}(40k, 12k)$, 20% with length $\mathcal{N}(10k, 3k)$	0, 25, 50, 75

Table 2: Lengths and arrival times of the job sets used in the three experiments.

In Figure 2 we compare how the SOCIAL and MIN-NUM relocators respond to a variation of the owner load of a machine. We use the synthetic grid, and the job set composed of two job types, namely C_0 and C_4 : jobs arrive uniformly in the interval $[0, 1000]$ s and their lengths are represented by a three-step discretization of a power law (more details described in section “Job set 1” of Table 2). The owner load of all the four machines is initially set to 0, but the owner load of machine m with speed vector S_3 increases to 0.95 at the time instant 1500 s, that is, it becomes *essentially* unavailable to the grid users. Plots in Figures 2(a), 2(b), and 2(c) show how jobs are distributed among machines with speed vectors S_0 , S_1 , and S_3 , respectively, under the SOCIAL relocator (the plot for machine S_2 is omitted because it is identical to that of S_1). A similar job distribution holds for the GREEDY relocator as well. Figures 2(d), 2(e), and 2(f) show job distribution under the MIN-NUM relocators in machines with speed vectors S_0 , S_1 , and S_3 , respectively. We notice that after the time instant 1500 s, the SOCIAL relocator begins to migrate jobs from m to other machines (with the GREEDY relocator exhibiting a similar behavior): indeed, the increasing number of jobs on the other machines is not due to new jobs, since no new job arrives after the time instant 1000 s. It is also interesting to note that jobs are distributed on machines according to their compositions: in particular, we observe that jobs of type C_4 (in cyan), dealing with large local data sets (i.e., $C_4[1] = 0.8$), are assigned by the allocator/relocator to m (i.e., S_3), which has the fastest disk, until the change in its owner load makes it essentially unavailable.

We now analyze a more general scenario using the AEOLUS grid. Jobs are described by composition vectors C_0 , C_1 , C_2 , and C_3 ; job lengths follow a four-step discrete discretization of a power law distribution, equal for all job types (see section “Job set 2” of Table 2). Since similar jobs in this environment are typically submitted in bursts, we consider four arrival times (0 s, 25 s, 50 s, and 75 s), and in each one only jobs described by the same composition vector arrive. Figure 3 provides the makespan (averaged on 5 simulation runs) of six

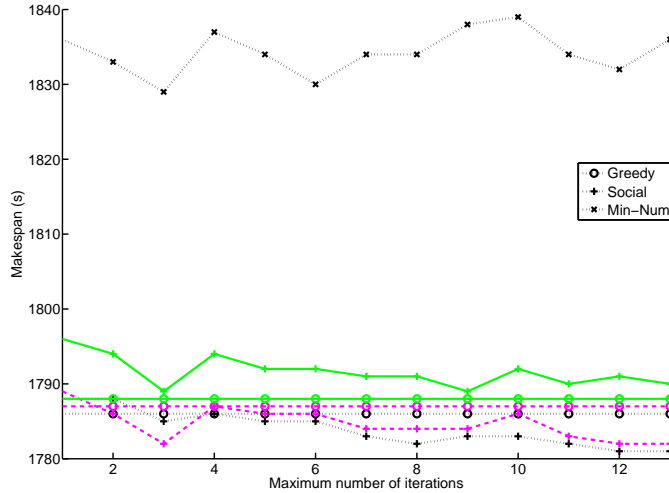


Figure 1: Behavior of the GREEDY, SOCIAL and MIN-NUM with varying N (on x -axis) and θ (cyan solid curves $\theta = 10\%$, magenta dashed curves $\theta = 5\%$, black dotted curves $\theta = 1\%$). θ is not defined for MIN-NUM.

allocation schemes (GREEDY, SOCIAL, MIN-NUM, each with and without the respective relocater), for any of the $4!$ orderings of job arrival times by job type. The mapping between permutation ID and the actual order of job types is provided in the Appendix (Table 4).

In the analyzed scenario, the GREEDY and SOCIAL relocators exhibit similar performances as noted before; in contrast, the MIN-NUM relocater experiences an average 10% performance loss. The experiment also provides evidence that, when relocation is not used for its high computational cost, the MIN-NUM should be avoided and the SOCIAL allocator is preferable to the GREEDY one: indeed, GREEDY wins over SOCIAL in 21% of the permutations with at most a 11% gap, while SOCIAL outperforms GREEDY on 79% of these instances, and the gap is more than 11% in 37% of the instances. It deserves to be noticed that SOCIAL beats GREEDY in particular in the first permutations, that is, when jobs with composition vector \mathcal{C}_3 are the last jobs submitted into the grid. The new jobs are allocated by the GREEDY allocator to machines with speed vector \mathcal{S}_0 , since these machines have a smaller number of assigned jobs. However, the affinities of these jobs decrease considerably (then, their execution times increase) and they cannot be migrated to other machines since the relocater is disabled. This problem is minimized in the SOCIAL allocator since the decrease in affinity of other jobs is taken into account in the score function.

We conclude this section with Figure 4, where we added to the previous scenario some owner load variations. Specifically, the owner load of the four machines characterized by speed vectors $\mathcal{S}_0, \mathcal{S}_6, \mathcal{S}_{26}, \mathcal{S}_{19}$ increases to 0.95 at time instants 0 s, 60 s, 100 s, 125 s, respectively. With relocation, performances are similar to those described above regarding Figure 3 since the relocations spread

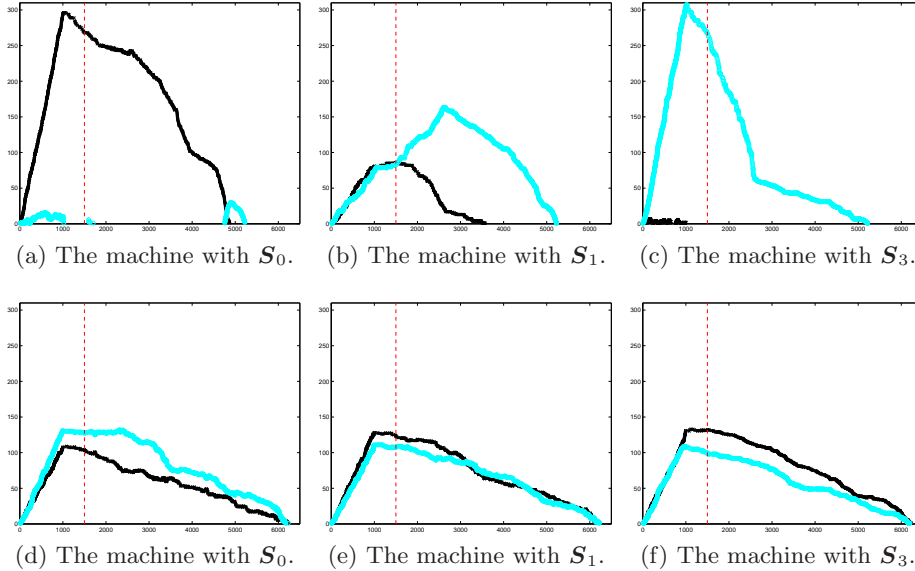


Figure 2: Job distribution on the synthetic grid on machines described by speed vectors S_0 , S_1 , and S_3 (we remind that $S_1 = S_2$): in (a), (b), and (c) under the SOCIAL relocater; in (d), (e), and (f) under the MIN-NUM relocater. In cyan jobs with composition vector C_4 ; in black jobs with composition vector C_0 . The x -axis reports the elapsed time (range $[0, 6500]$ s), the y -axis the number of jobs on the machine (range $[0, 310]$).

jobs from the four “nearly unavailable” machines to the remaining 66 machines. In contrast, disabling relocations yields huge makespans, in particular under the MIN-NUM allocator since it does not take into account the owner load of a machine.

4 Conclusions and Future Work

In this paper, we have proposed a new framework for resource allocation in desktop grid environments based on the idea of performing job assignments to the machines which best meet the computational requirements of the jobs. Within this framework, we have developed and compared two different allocation schemes, which attempt to minimize the overall system makespan, even without knowing the actual durations of the jobs submitted to the system. We have argued that our strategy results in a proper, fair, and balanced allocation of the jobs processed by the grid, and this translates into good results in terms of completion time of the jobs. The proposed framework can be extended in several ways: first, by introducing the concept of *domain* of a job, that is, allowing a job to choose, on the basis of their hardware or software capabilities (e.g., CPU architecture, amount of RAM and disk space, operating system installed, available software libraries), the subset of machines on which its computation can be carried out (notice that this simple extension would add a new combinatorial dimension to the problem, since both the allocation and the relocation

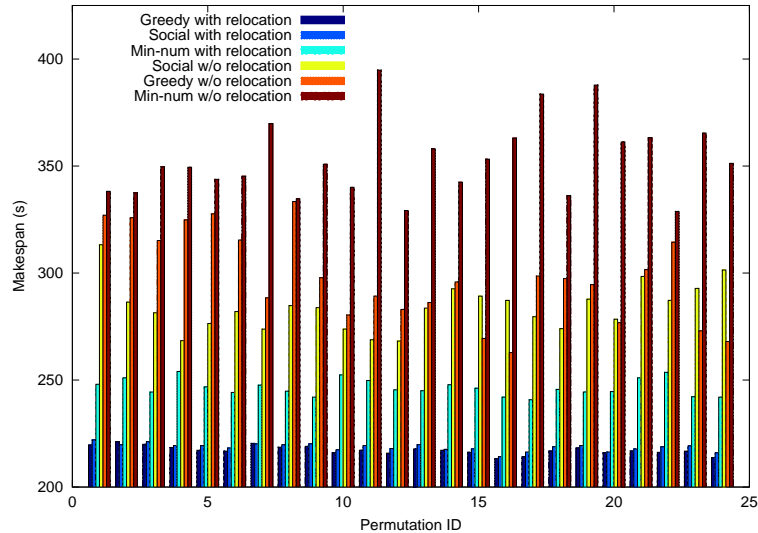


Figure 3: Makespan on the AEOLUS grid of six allocation schemes (GREEDY, SOCIAL, MIN-NUM, each with and without the respective rebalancing), for each of the $4!$ permutations of job arrival times by job type (permutation IDs are listed in Table 4).

choices would have to deal with intersecting domains); then, it would be useful, for robustness and scalability purposes, to implement our scheduler in a distributed fashion; finally, this approach deserves to be implemented in a real grid environment (such as the AEOLUS testbed), and possibly to compare its performances with other state-of-the-art resource brokering systems and time-driven scheduling strategies.

Acknowledgments. The authors would like to thank Andrea Pietracaprina and Geppino Pucci for helpful discussions and comments, and Joachim Gehweiler for his help on the AEOLUS testbed.

References

1. D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *Proc. of the 14th Int. Parallel & Distributed Processing Symp.*, pages 520–528, 2000.
2. AEOLUS testbed website. <http://aeolus.cs.upb.de>.
3. F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. M. Figueira, J. Hayes, G. Obertelli, J. M. Schopf, G. Shao, S. Smallen, N. T. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.
4. P. Bertasi, M. Bianco, A. Pietracaprina, and G. Pucci. Obtaining performance measures through microbenchmarking in a peer-to-peer overlay computer. *Int. J. of Computational Intelligence Research*, 4(1):1–8, 2008.
5. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Proc. of the 9th Heterogeneous Computing Workshop*, pages 349–363, 2000.

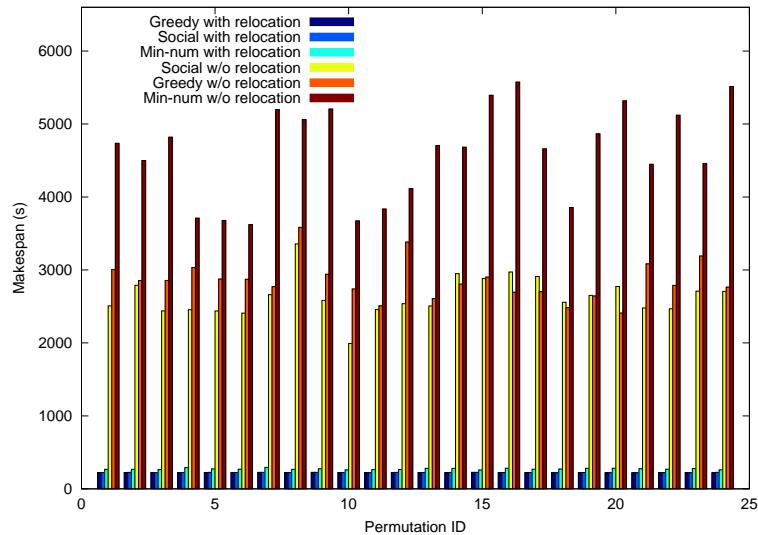


Figure 4: Makespan on the AEOLUS grid of six allocation schemes (GREEDY, SOCIAL, MIN-NUM, each with and without the respective rebalancing), for each of the $4!$ permutations of job arrival times by job type. The owner load of the four machines characterized by speed vectors S_0, S_6, S_{26}, S_{19} increases to 0.95 at time instants 0s, 60s, 100s, 125s, respectively (permutation IDs are listed in Table 4).

6. Parallel Workloads Archive <http://www.cs.huji.ac.il/labs/parallel/workload>.
7. D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proc. of the 12th Int. Parallel Processing Symp. / 9th Symp. on Parallel and Distributed Processing*, pages 542–546, 1998.
8. I. Foster and C. Kesselman. Globus: A meta-computing infrastructure toolkit. *Int. J. of Supercomputer Applications*, 11(2):115–128, 1997.
9. I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2nd edition, 2003.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
11. M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Computer Systems*, 15(3):253–285, 1997.
12. O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. of the ACM*, 24(2):280–289, 1977.
13. B. B. Khoo, B. Veeravalli, T. Hung, and C. W. Simon See. A multi-dimensional scheduling scheme in a grid computing environment. *J. Parallel and Distributed Computing*, 67(6):659–673, 2007.
14. C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Job Scheduling Strategies for Parallel Processing*, pages 253–263, 2004.
15. Y. C. Lee and A. Y. Zomaya. Practical scheduling of bag-of-tasks applications on grids with dynamic resilience. *IEEE Trans. on Computers*, 56(6):815–825, 2007.
16. W. Leland and T. J. Ott. Load-balancing heuristics and process behavior. *ACM SIGMETRICS Performance Evaluation Review*, 14(1):54–69, 1986.

17. J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.
18. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – a hunter of idle workstations. In *Proc. of the 8th Int. Conf. on Distr. Computing Systems*, pages 104–111, 1988.
19. C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *Proc. of the 11th IEEE Int. Symp. on High Performance Distributed Computing*, pages 63–72, 2002.
20. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *J. Parallel and Distributed Computing*, 59(2):107–131, 1999.
21. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of the 7th IEEE Int. Symp. on High Performance Distributed Computing*, pages 140–146, 1998.

Appendix

In this appendix we provide the speed vectors of the machines in the AEOLUS grid, listed in Table 3, while Table 4 shows the mapping between permutation IDs and arrival times, used in Figures 3 and 4.

Speed vector	CPU (GHz)	Disk (MB/s)	Network (KB/s)	Speed vector	CPU (GHz)	Disk (MB/s)	Network (KB/s)
S_0	0.70	50	500	S_{20}	2.20	100	500
S_1	0.80	50	500	S_{21}	2.40	100	500
S_2	0.87	50	500	S_{22}	2.40	100	1000
S_3	0.90	50	500	$S_{23} - S_{24}$	2.53	90	1000
$S_4 - S_5$	0.93	50	500	S_{25}	2.60	90	1000
S_6	1.00	60	500	$S_{26} - S_{55}$	2.66	105	1000
S_7	1.30	60	500	$S_{56} - S_{61}$	2.80	85	500
S_8	1.40	60	500	$S_{62} - S_{63}$	2.83	100	500
S_9	1.67	75	500	$S_{64} - S_{65}$	3.00	80	500
$S_{10} - S_{15}$	1.70	70	500	$S_{66} - S_{67}$	3.10	80	500
S_{16}	1.80	70	500	$S_{68} - S_{69}$	3.20	80	500
$S_{17} - S_{19}$	2.00	100	500				

Table 3: Speed vectors of the 70 machines in the AEOLUS grid.

ID	0 s	25 s	50 s	75 s	ID	0 s	25 s	50 s	75 s	ID	0 s	25 s	50 s	75 s
1	C_0	C_1	C_2	C_3	9	C_0	C_3	C_1	C_2	17	C_2	C_3	C_0	C_1
2	C_1	C_0	C_2	C_3	10	C_3	C_0	C_1	C_2	18	C_3	C_2	C_0	C_1
3	C_0	C_2	C_1	C_3	11	C_3	C_1	C_0	C_2	19	C_3	C_1	C_2	C_0
4	C_2	C_0	C_1	C_3	12	C_1	C_3	C_0	C_2	20	C_1	C_3	C_2	C_0
5	C_2	C_1	C_0	C_3	13	C_0	C_3	C_2	C_1	21	C_3	C_2	C_1	C_0
6	C_1	C_2	C_0	C_3	14	C_3	C_0	C_2	C_1	22	C_2	C_3	C_1	C_0
7	C_0	C_1	C_3	C_2	15	C_0	C_2	C_3	C_1	23	C_2	C_1	C_3	C_0
8	C_1	C_0	C_3	C_2	16	C_2	C_0	C_3	C_1	24	C_1	C_2	C_3	C_0

Table 4: Arrival time of each job type for each permutation ID (see Figures 3 and 4).