# Fair Scheduling Algorithms in Grids

Nikolaos D. Doulamis, *Member*, *IEEE*, Anastasios D. Doulamis, *Member*, *IEEE*,
Emmanouel A. Varvarigos, and Theodora A. Varvarigou, *Member*, *IEEE*

**Abstract**—In this paper, we propose a new algorithm for fair scheduling, and we compare it to other scheduling schemes such as the Earliest Deadline First (EDF) and the First Come First Served (FCFS) schemes. Our algorithm uses a max-min fair sharing approach for providing fair access to users. When there is no shortage of resources, the algorithm assigns to each task enough computational power for it to finish within its deadline. When there is congestion, the main idea is to fairly reduce the CPU rates assigned to the tasks so that the share of resources that each user gets is proportional to the user's weight. The weight of a user may be defined as the user's contribution to the infrastructure or the price he is willing to pay for services or any other socioeconomic consideration. In our algorithms, all tasks whose requirements are lower than their fair share CPU rate are served at their demanded CPU rates. However, the CPU rates of tasks whose requirements are larger than their fair share CPU rate are reduced to fit the total available computational capacity in a fair manner. Three different versions of fair scheduling are adopted in this paper: the Simple Fair Task Order (SFTO), which schedules the tasks according to their respective fair completion times, the Adjusted Fair Task Order (AFTO), which refines the SFTO policy by ordering the tasks using the adjusted fair completion time, and the Max-Min Fair Share (MMFS) scheduling policy, which simultaneously addresses the problem of finding a fair task order and assigning a processor to each task based on a max-min fair sharing policy. Experimental results and comparisons with traditional scheduling schemes such as the EDF and the FCFS are presented using three different error criteria. Validation of the simulations using real experiments of tasks generated from 3D image-rendering processes is also provided. The three proposed scheduling schemes can be integrated into existing Grid computing architectures.

**Index Terms**—Grid computing, fair grid scheduling.

## 1 Introduction

Scheduling and resource management are important in optimizing multiprocessor Grid resource allocation and determining its ability to deliver the negotiated Quality-of-Service (QoS) requirements [1], [2]. This need has been confirmed by the Global Grid Forum (GGF) in the special working group dealing with the area of scheduling and resource management for Grid computing [3]. The resource manager receives information about the job characteristics and determines *when and on which* processor each job will execute.

Though Grid computing has been exaggerated by the international community, it is now cooling down. Nowadays, the major issue concerns the effective system integration by efficiently utilizing the existing tools with research results that will make Grid computing applicable to many commercial scenarios. Toward this direction, scheduling and resource allocation schemes play a determinant role. In a scheduling scheme, however, meeting the requirements of one user should not be achieved by sacrificing the requirements of another user. When the desired users' requirements cannot be achieved, the degradation should be graceful and fair to all users. As the tasks' requirements, we refer to the tasks' deadlines, workload, and the time that a task is ready to be executed on a processor. This naturally leads to the need of *congestion control* and the associated notion of *fairness* issues that we address in this paper. Fairness is important because it is inherent in the notion of sharing, which is the raison d'etre of the Grid.

Several computing toolkits and systems have been developed to guarantee the QoS requirements of tasks in a Grid computing architecture. The most well-known toolkit for Grid computing is Globus [4]. Globus addresses a wide range of metacomputing issues including heterogeneous environments. Development, implementation, and evaluation of mechanisms that support High Throughtput Computing (HTC) on a large collection of distributively owned computing resources are also addressed in the framework of Condor project [5]. The Grid version of Condor, called Condor-G, uses the Globus toolkit to manage Grid jobs [6], [7]. Condor has been designed to run jobs within a single administrative domain. On the other hand, the Globus toolkit has been designed to run jobs across many administrative domains. Condor-G combines the strengths of both. Condor-G introduces grid scheduler and manager to allow full-featured queuing services, credential manager, and fault-tolerance issues. An object-oriented parallel processing distributed computing enhanced with security capabilities is the Legion system. Legion will provide a single, coherent virtual machine that addresses scalability, programming ease, fault tolerance, site autonomy, and security [8].

- *N.D. Doulamis and T.A. Varvarigou are with the Department of Electrical and Computer Engineering, National Technical University of Athens, Greece. E-mail: ndoulam@cs.ntua.gr, dora@telecom.ntua.gr.*
- *A.D. Doulamis is with the Technical University of Crete, Chania, Greece. E-mail: adoulam@cs.ntua.gr.*
- *E.A. Varvarigos is with the Department of Computer Engineering and Informatics, University of Patras, Greece. E-mail: manos@ceid.upatras.gr.*

*Manuscript received 1 Aug. 2004; revised 2 Jan. 2006; accepted 28 Sept. 2006; published online 8 Jan. 2007.*
*Recommended for acceptance by X. Zhang.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0191-0804.*
*Digital Object Identifier no. 10.1109/TPDS.2007.1053.*

1045-9219/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

The Nimrod-G [9] is a Grid aware version of the Nimrod tool [10]. Nimrod provides a simple declarative parametric modeling language for expressing a parametric experiment. The domain experts can easily create a plan for a parametric computing (task farming) and use the Nimrod runtime system to submit, run, and collect the results from multiple computers (cluster nodes) [11]. The Grid Application Development Software (GrADS) tool aims at simplifying distributed heterogeneous computing in the same way that the World Wide Web simplifies information sharing over the Internet [12]. The GrADS environment supports scheduling algorithms at the application level and metalevel [13]. In the application level, scheduling of a single application is performed by minimizing the application execution time of a set of potentially shared resources. Instead, in the metascheduling level, many applications are considered at once to improve the overall system performance.

In the G-commerce architecture, a method of a dynamic Grid resource allocation is adopted using notions of market economy [14]. Two categories are considered in the framework of this architecture: the *commodities* and the *auctions*. Modeling the Grid as a commodities market is natural since the Grid strives to allow applications to treat disparate resources as interchangeable commodities. On the other hand, auctions require little in the way of global price information, and they are easy to implement in a distributed setting.

The Queue Bank or Quantum Bank (QBank) system is a CPU allocation bank that consists of a set of routines, a server daemon, client administration tools, and commands able to manage and control the allocation of the CPU resources on a supercomputer. The QBank was developed for:

1. controlling and managing CPU resources allocated to tasks or users,
2. applying different billing rates according to the policies adopted, tasks, and resource types,
3. providing balance and usage feedback to users and administrators, and
4. preventing resource exhaustion when underspent tasks simultaneously claim allocation fulfillment [15].

A fully distributed view of the Grid usage accounting system and a methodology for allocating Grid computational resources for use on a Grid is presented in the work performed by the distributed accounting working group (DAWG) of the GGF [16]. In particular, a usage economy and/or methods for resource exchange are defined along with implementation standards that minimize and compartmentalize the tasks required for a site to be participated in Grid accounting.

Apart from the above mentioned Grid infrastructures, several works have been reported in the literature dealing with scheduling and resource allocation. The most well-known scheduling algorithm is the Earliest Deadline First (EDF) [17], which assigns the highest priority to the task with the most imminent deadline. Another scheduling approach is the Slack Time algorithm, also referred as Least Laxity First (LLF) [18], [19], where the tasks are selected for execution in order of nondecreasing slack time, defined as the difference between the task's relative deadline and its remaining computational time.

However, the aforementioned schemes have been designed for a single processor. Even though these schemes can be extended in a natural way to the case of a multiprocessor environment to determine the order in which the tasks are considered for assignment to processors, they cannot be used to determine the specific processor on which the selected tasks are assigned to. A simple rule to determine the processor on which a task is executed is the Earliest Start Time (EST) rule. The EST is the earliest time that a task can start its execution. Another popular rule is the Minimum Processing Time First rule (MPTF), where the processor giving the minimum processing time is selected. For a multiprocessor system, the authors in [21] have shown that heuristic schemes that takes into account both the task deadline and EST better performs than the EDF, LLF, and MPTF algorithms. The authors in [22] also proposed several heuristic scheduling algorithms for the multiprocessor case.

The previously mentioned scheduling algorithms assume that the tasks are nonpreemptable. A task is said to be nonpreemptable if, once it starts execution on a processor, it has to be completed on that processor, and once it starts execution, it cannot be interrupted by other tasks and resume execution later. Scheduling algorithms dealing with preemptable tasks have also been reported in the literature [23], [24], [25], [26]. In this case, it is assumed that each task can be divided into smaller units, each of which is executed independently. The ability to feasibly schedule preemptable tasks is always higher than the ability to feasibly schedule corresponding nonpreemptable tasks. However, this increase in schedulability is obtained at the expense of a higher implementation overhead. To address this difficulty, task parallelization can be performed as an intermediate solution, which tries to meet the conflicting requirements of schedulability and overhead.

Other scheduling schemes are oriented for Grid computing. In [27], an extension of the scheduling algorithms of the GrADS tool is discussed by 1) introducing more sophisticated clustering and data mining schemes, 2) reducing the computational complexity, and 3) providing single-site scheduling in case of invalidation of multisite resource selection. The scheduling objective in [28] is to minimize the total completion time of the tasks. Since minimization of task completion time in a multiprocessor scheduling environment is an NP-hard problem, scheduling heuristics are discussed and compared with each other in this work. Genetic algorithm methods are presented in [29] and [30] for minimizing the total task completion time. The algorithms model the scheduling process as a *genetic evolution* and estimate at which Grid resource a task should be assigned for execution so that the completion time is minimized.

A survey evaluation of scheduling algorithms is presented in [31]. Stochastic evaluation of fair scheduling algorithms is also presented in [32], where networking issues are discussed. Finally, evaluation of different scheduling mechanisms for Grid computing is also presented in [33], such as the First Come First Served (FCFS), the Largest Time First (LTF), the Largest Cost First (LCF), the Largest Job First (LJF), the Largest Machine First (LMF),

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

the Smallest Machine First (SMF), and the Minimum Effective Execution Time (MEET).

A drawback of the previously mentioned approaches is that scheduling is performed without taking into account *fair considerations*. For example, the works in [28], [29], and [30] try to minimize the total completion time by dropping overdemanded tasks (for example, tasks of high workload and short deadlines), which is not a fair policy. Instead, in this paper, a fair scheduling policy is introduced based on the max-min fair scheduling scheme. The Generalized Processor Sharing scheme (GPS) has been proposed for fair scheduling over packet switched networks [34]. The GPS scheme provably provides guarantees on the delay and bandwidth of a session in a network of switches but is hard to implement. The GPS scheme is emulated in practice using the Weighted Fair Queuing (WFQ) algorithm [35], which exploits concepts of the max-min fair sharing scheme [36]. GPS-based algorithms are widely implemented in the Internet and mobile communications today.

In this paper, three new scheduling algorithms are proposed suitable for Grid computing. All scheduling policies are based on the max-min fair sharing scheme. The first, called Simple Fair Task Order (SFTO) algorithm, schedules the tasks according to their respective fair completion times and then assigns each of them to the appropriate processor using a modified Earliest Completion Time (ECT)-based policy. The second, called Adjusted Fair Task Order (AFTO), refines the SFTO policy by ordering the tasks using the adjusted fair completion times, resulting in more fair treatment of the jobs. Finally, for the third scheme, we present the Max-Min Fair Share (MMFS) scheduling algorithm, simultaneously address the problem of finding a fair task order, and assign a processor to each task based on a max-min fair sharing policy. Experimental results and comparisons with traditional scheduling schemes such as the EDF and the FCFS are presented using three different error criteria. The simulations have been conducted using a large number of processors, ranging from 50 to 1,000, using a very large number of tasks 2,500 to 6,500 of varying sizes (workload) and deadlines. The simulations are also performed for varying capacities of the processors using either the symmetric case (almost all processors follow the same capacity) or asymmetric case (high deviation in the processor capacity) into groups or distributions of high standard deviation values. Validation of the simulations using real submitted tasks as derived from 3D image-rendering applications is also examined. The experiments are conducted *in a real multiprocessor Grid cluster* implemented in the framework of the GRID Resources for Industrial Applications (GRIA) and GRIDLAB European Union (EU)-funded projects. As a result, the algorithms are applicable for large-scale computing systems embedded with multiple processors. The proposed scheduling algorithms can be also integrated in any other existing Grid computing system to improve its performance as far as the task allocation to the available processors is concerned.

The remainder of the paper is organized as follows: In Section 2, we discuss the relation between the scheduling policy and the adopted charging policy, an issue that is closely related to the commercial exploitation of Grid computing. In Section 3, we introduce some basic notation, whereas in Section 4, we present urgency-based scheduling schemes and the modified ECT policy used for processor assignment. Starting with Section 5, we turn our attention to scheduling algorithms that take fairness into account, and we introduce the SFTO and AFTO schemes. In Section 6, we present the MMFS scheduling algorithm, which simultaneously addresses the problem of finding a fair task order and assigning a processor to each task, whereas experimental results and comparisons with traditional scheduling schemes are presented in Section 7. Finally, Section 8 concludes the paper.

## 2 Scheduling Evaluation and Its Relation to the Charging Policy

Evaluating the efficiency of a scheduling algorithm depends on the utility function that we seek to optimize, which in turn depends on technoeconomic criteria. For example, a scheduling algorithm that maximizes the number of tasks served by the Grid tends to favor tasks of low workload at the expense of tasks of heavy workload. The opposite is true when the evaluation criterion used is the total workload served, since, in that case, there is a tendency to reject tasks of low workload in favor of tasks of heavy workload. A fair scheduling algorithm, however, should not favor tasks of specific characteristics (for example, high or low workload) against others.

In a scheduling problem, the goal is to appropriately assign all the tasks that are requesting service to the available processors so that the time constraints are satisfied. The time constraints of a task are the task's deadline (which is the time by which it is desirable for it to complete execution) and the task's earliest starting time on each processor (which is the earliest time at which the task can start execution at that processor; it takes into account the communication delay incurred for transferring the task at that processor and the current load of that processor). This problem may have zero, one, or many feasible solutions. Often, finding a single feasible schedule may not be sufficient. In some cases, the goal may be to find the optimal schedule among all feasible schedules, according to a desired optimality criterion. In other cases, a feasible solution may not exist, in the sense that some tasks cannot be scheduled to meet their respective deadlines. In this case, we need criteria to select in a "fair" way the tasks that are rejected and the tasks that receive a degraded QoS. The fairness of a solution depends on the adopted charging policy of the system.

A common measure for evaluating the scheduling performance is the *success ratio*, defined as the ratio of the number of tasks that are feasibly scheduled (that is, the tasks whose time constraints are met) over the total number of tasks requesting service. This measure treats all tasks equally, regardless of their workload, and it does not take into account the users' contribution to the Grid infrastructure or the price that a user pays for the service he receives. Another performance measure is the *total workload* of all feasibly scheduled tasks. Here, the charging policy is implemented per workload unit, and it is more beneficial to serve tasks of heavy workload than tasks of low workload. In such a case, the fees charged are proportional to the customer task workload, so it is preferable to serve a few customers who are willing to pay a lot, rather than a lot of customers who are willing to pay only little for their services. Such a policy is similar to the one used in traditional telephone networks.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

The objective of the algorithms that we will present in Sections 5 and 6 for scheduling tasks with deadlines on resources is to optimize a *fairness* criterion. A fair scheduler tries to meet the different requirements of the users. When this is not possible, the resources are allocated so that the degradation in some requirements perceived by each user is done in a (weighted) fair way. For example, tasks submitted by users who are willing to pay a higher fee or who contribute more to the Grid infrastructure are treated more favorably (in a measurable way) than those of other users.

## 3 Notation and Problem Formulation

We let $N$ be the number of tasks that have to be scheduled. We define the workload $w_i$ of task $T_i$, $i = 1, 2, \ldots, N$, as the duration of the task when executed on a processor of unit computation capacity. The task workloads are assumed to be known a priori to the scheduler and are provided by a prediction mechanism such as script discovery algorithms, databases containing statistical data on previous runs of similar tasks, and so forth. An algorithm for workload prediction of 3D rendering in a Grid architecture is presented in one of our earlier works in [37]. We assume that the tasks are nonpreemptable, so that when they start execution on a machine, they run continuously on that machine until completion. We also assume that time sharing is not available and a task served on a processor occupies 100 percent of the processor capacity.

We assume a multiprocessor system of $M$ processors and that the computation capacity of processor $j$ is equal to $c_j$ units of capacity. (The computation capacity of a processor is the available capacity of the processor, and it does not include capacity occupied by local tasks.) The *total computation capacity* $C$ of the Grid is defined as

$$C = \sum_{j=1}^{M} c_j. \tag{1}$$

Let $d_{ij}$ be the communication delay between user $i$ and processor $j$. More precisely, $d_{ij}$ is (an estimate of) the time that elapses between the time a decision is made by the resource manager to assign task $T_i$ to processor $j$ and the arrival of all files necessary to run task $T_i$ to processor $j$.

Each task $T_i$ is characterized by a deadline $D_i$ that defines the time by which it is *desirable* for the task to complete execution. In our formulation, $D_i$ is not necessarily a hard deadline. In case of congestion, the scheduler may not assign sufficient resources to the task to complete execution before the deadline. In that case, the user may choose not to execute the task, as may be the case when he/she expects the results to be outdated or not useful by the time they are provided. We use $D_i$ together with the estimated task workload $w_i$ and the communication delays $d_{ij}$ to obtain estimates of the computation capacity that task $T_i$ would have to reserve to meet its deadline if assigned to processor $j$. If the deadline constraints of all tasks cannot be met, our target is that a schedule that is feasible with respect to all other constraints is still returned, and the amounts of time by which the tasks miss their respective deadlines is determined in a fair way.

We let $\gamma_j$ be the estimated completion time of the tasks that are already running on or already scheduled on processor $j$. $\gamma_j$ is equal to zero (that is, the present time) when no task has been allocated to processor $j$ at the time a task assignment is about to be made; otherwise, $\gamma_j$ corresponds to the remaining time until the completion of the tasks that are already allocated to processor $j$. We define the *earliest starting time* of task $T_i$ on processor $j$ as

$$\delta_{ij} = \max\{d_{ij}, \gamma_j\}. \tag{2}$$

$\delta_{ij}$ is the earliest time at which it is feasible for task $T_i$ to start execution on processor $j$. We define the average of the earliest starting times of task $T_i$ over all the $M$ available processors as

$$\delta_i = \frac{\sum_{j=1}^{M} \delta_{ij} c_j}{\sum_{j=1}^{M} c_j}. \tag{3}$$

We will refer to $\delta_i$ as the *grid access delay* for task $T_i$, and it can be viewed as the (weighted) mean delay required for task $T_i$ to access the total grid capacity $C = \sum_{j=1}^{M} c_j$. Since in a Grid computation power is distributed, $\delta_i$ plays a role reminiscent of that of the (mean) memory access time in uniprocessor computers.

In the fair scheduling algorithm that we will propose in Section 5, the *demanded computation rate* $X_i$ of a task $T_i$ will play an important role and is defined as

$$X_i = \frac{w_i}{D_i - \delta_i}. \tag{4}$$

$X_i$ can be viewed as the computation capacity that the Grid should allocate to task $T_i$ for it to finish just before its requested deadline $D_i$ if the allocated computation capacity could be accessed at the mean access delay $\delta_i$. As we will see later, the computation rate allocated to a task may have to be smaller than its demanded rate $X_i$. This may happen if more jobs request service than the Grid can support (congestion), in which case, some or all of the jobs may have to miss their deadline. The fair scheduling algorithms of Sections 5 and 6 attempt to degrade the tasks' rates in a fair way.

The scheduling algorithms that we will propose (except for the MMFS algorithm proposed in Section 6) consist of two phases. In the first phase, we determine the order in which tasks will be considered for assignment to processors (the "queuing order" phase), and in the second phase, we determine the processor on which each task is scheduled (the "processor assignment" phase).

### 3.1 Arrival Model

In this section, we describe the arrival model adopted in the experimental simulations in this paper. Initially, we define the normalized load of the grid infrastructure as the ratio of the tasks' demanded computational rates $X_i$ over the total processor capacity $C$ offered by the grid infrastructure:

$$\rho = \frac{\sum_{i=1}^{N} X_i}{C}. \tag{5}$$

From (5), it is clear that a grid with load $\rho$ is able to serve, on the average, $N$ tasks of workload $w_i$, deadlines $D_i$, and

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

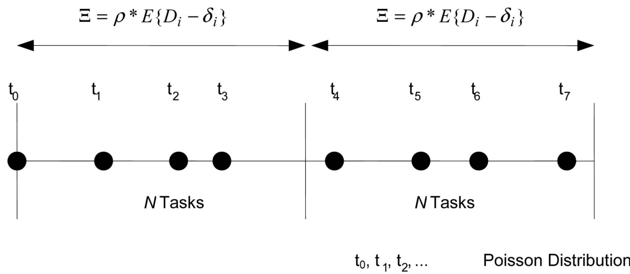



Fig. 1. An illustrative example of the adopted arrival model.

ready times $\delta_i$ within a time interval of $\Xi = \rho * E\{D_i - \delta_i\}$, where the $E\{\cdot\}$ denotes the expectation operator. As a result, arrival of an average number of $N$ tasks within a time interval of $\Xi = \rho * E\{D_i - \delta_i\}$ does not change the load $\rho$ of the Grid.

In the arrival model adopted, in this paper, we assume that the $N$ tasks arrive in the Grid into groups of $N/\beta$ tasks. We also assume that the probability of each group of $N/\beta$ tasks to arrive in the Grid during an interval $(0, t_0)$ of duration $t_0$ follows the Poisson distribution:

$$P(t = t_o) = \frac{e^{-\lambda}\lambda^{t_o}}{t_o!} \tag{6}$$

with parameter $\lambda$ equals $\lambda^{-1} = \beta \cdot \Xi$. At the time $t = t_0$, the resource management is activated for scheduling the $N/\beta$ tasks that have arrived by time $t = t_0$. The scheduling policies adopted are described in the rest of this paper. In our experiments, $\beta$ equals 10. Fig. 1 presents an illustrative example of the adopted arrival model.

## 4 Earliest Deadline First and Earliest Completion Time Rules

The most widely used urgency-based scheduling scheme is the EDF method, also known as the deadline-driven rule. This method dictates that, at any point, the system must assign the highest priority to the task with the most imminent deadline. The most urgent tasks (that is, the task with the earliest deadline) are served first, followed by the remaining tasks according to their urgency.

The EDF rule answers only the "queuing order" question, but it does not determine the processor where the selected task is assigned. To answer the "processor assignment" question, the ECT technique presented next can be used. The EDF/ECT algorithm is also identical to the Horizon scheduling used in burst switched networks [38].

If task $T_i$ starts execution on processor $j$ at the earliest starting time $\delta_{ij}$, its completion time will be $\delta_{ij} + w_{ij}$, where $w_{ij} = w_i/c_j$ is the execution time of task $T_i$ on processor $j$. (Recall our assumption that each task occupies 100 percent of a processor's capacity when executed; in this way, tasks are executed in the earliest possible time.) Among the $M$ available processors, the ECT rule selects the one that minimizes the following quantity:

$$\hat{j} = \underset{j \in \{1,\cdots,M\}}{\arg\min} \{\delta_{ij} + w_{ij}\}. \tag{7}$$

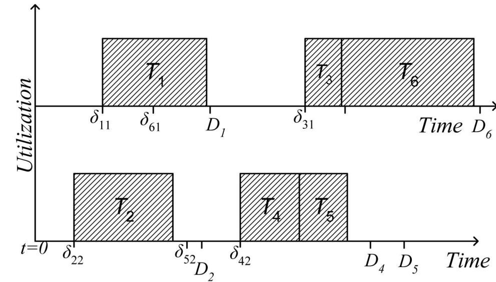



Fig. 2. An example of the EDF/ECT algorithm for the case $\gamma_j$ is defined as the processor release time. In this figure, we assume that all tasks $T_i$, $i = 1, 2, \ldots, 6$, request service at time $t = 0$, the processors have equal computational capacity $(c_1 = c_2)$, and both processors are initially idle. We also assume that $D_1 < D_2 < \cdots < D_6$ and $\delta_{i1} = \delta_{i2}$. The task $T_1$ of the earliest deadline $D_1$ is first assigned for execution on processor 1 (processor 1 is chosen randomly in this case, since there is a tie). Task $T_2$ is then assigned for execution on processor 2 (since it is the processor that yields the ECT). In a similar way, we assign the remaining tasks.

The earliest starting time $\delta_{ij}$ depends through (2) on the time $\gamma_j$ at which the last task already allocated to processor $j$ is expected to complete service.

A note regarding the way $\gamma_j$ is defined is necessary here. One way to define $\gamma_j$ is to define it as the *processor release time*, that is, the time at which all tasks already scheduled on this processor finish their execution. Fig. 2 illustrates a scheduling scenario in which 1) the *task queuing order* is selected using the EDF algorithm, 2) the *processor assignment* is selected using the ECT approach, and 3) $\gamma_j$ is defined as the processor release time. In this example, we assume that all tasks arrive at time $t = 0$.

Defining $\gamma_j$ as the processor release time makes it easy to compute and independent of the task that is about to be scheduled, but it has the drawback that gaps in the utilization of a processor are created (for example, the gap between tasks $T_1$ and $T_3$ in Fig. 1), resulting in a waste of processor capacity. An obvious way to overcome this problem is to examine the capacity utilization gaps and, in case a task fits within a capacity gap, assign the task to the corresponding time interval. Among all candidate time intervals, the one that provides the ECT is selected. Fig. 3 shows how the schedule for the example given in Fig. 1 is improved by exploiting capacity gaps. The completion times of tasks $T_5$ and $T_6$ are shorter than those obtained in Fig. 1. The gap filling version of the algorithm is very similar to the Latest Available Unused Channel with Void Filling (LAUC-VF) adopted in burst switching [39].

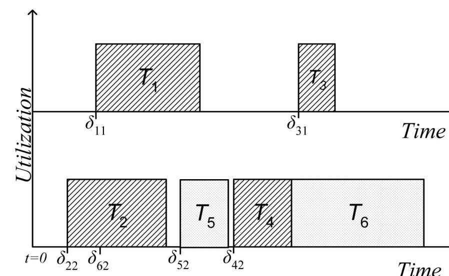



Fig. 3. An example of the EDF/ECT algorithm that exploits processor utilization gaps.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

TABLE 1
An Example of the Nonweighted Max-Min Fair Sharing Algorithm If the Overall Processor Capacity is 30

| Demanded Rates | Max-Min Fair Sharing (First iteration) | Max-Min Fair Sharing (Second iteration) | Fair Rates |
|---|---|---|---|
| 10 | 7.5 | 10 | 10 |
| 3 | 3 | 3 | 3 |
| 5 | 5 | 5 | 5 |
| 15 | 7.5 | 11 | 12 |
| **Residue** | 30-23=7 | 1 | 0 |

## 5 Fair Scheduling

The scheduling algorithms described in Section 4 do not adequately address congestion, and they do not take fairness considerations into account. For example, tasks with relative urgency (with the EDF rule) or tasks that have small workload (with the ECT rule) are favored against the remaining tasks. With the ECT rule, tasks that have long execution time have a higher probability of missing their deadline even if they have a late deadline. Also, with the EDF rule, a task with a late deadline is given low priority until its deadline approaches, giving no incentive to the user to specify an honest deadline (especially in the absence of any pricing mechanism). To overcome these difficulties, we propose in this section an alternative approach, where the tasks requesting service are queued for scheduling according to what we call their *fair completion times*. The fair completion time of a task is found by first estimating its *fair task rates* using a max-min fair sharing algorithm as described in the following section. It should be mentioned that the algorithms proposed in this paper are oriented for large-scale computing systems in which multiple processors are taken into account.

### 5.1 Estimation of the Task Fair Rates

#### 5.1.1 Ideal Nonweighted Max-Min Fair Sharing of Grid Resources

Intuitively, in max-min fair sharing, all users are given an equal share of the total resources, unless some of them do not need their whole share, in which case, their unused share is divided equally among the remaining "bigger" users in a recursive way. In other words, in the max-min fair sharing scheme, small demanded computation rates $X_i$ get all the computation power they require, whereas larger rates share leftovers.

The idea of the max-min fair sharing is explained by the following example, where four tasks request service with rates 10, 8, 5, and 15, respectively, in a multiple processor architecture. The iterations involved are shown in Table 1. Let us assume that the total offered processor capacity equals 30 units. The total demanded rate of the tasks equals $10 + 3 + 5 + 15 = 32 \text{ units}$, which is greater than the total offered processor capacity. As a result, the max-min fair sharing algorithm reduces the task rates in a fair way so that the demanded task rates equal the total offered processor capacity. Since all tasks are of equal importance, the algorithm initially divides the 30 units of the total processor capacity into four equal parts of $30/4 = 7.5 \text{ units}$. The second and the third task request service less than 7.5 units (3 and 5, respectively) and, thus, they get the rate they request. Instead, the first and fourth task demand rate more than 7.5 units (10 and 15, respectively) and, therefore, at the first iteration of the algorithm, a rate of 7.5 units is assigned to them. Consequently, at the end of the first iteration, a residue of $30 - 23(7.5 + 3 + 5 + 7.5) = 7$ units is obtained. In the second iteration, the residue of seven units is equally shared among the first and fourth task, whose actual rates are less than the demanded ones, so that each of the two tasks can get an additional rate of $7/2 = 3.5 \text{ units}$. Since, however, the first task request service less than $11(7.5 + 3.5) \text{ units}$, it gets the rate it requests, that is, 10 units. On the contrary, the fourth task gets a rate of 11 units and the end of second iteration, and a residue of one unit is accomplished. This residue is then shared to the fourth task whose rate is less than the demanded one in the third iteration of the algorithm. At the end of the nonweighted max-min fair sharing algorithms, the non-adjusted fair computational rates $r_i$ of tasks $T_i$ are computed. A graphical conceptualization of the aforementioned example is depicted in Fig. 4.

More details about the max-min fair sharing algorithm can be found in Appendix A.

#### 5.1.2 Ideal Weighted Max-Min Fair Sharing of the Grid Resources

We now consider the case where users have different priorities. More specifically, we assume that each task $T_i$ is assigned an integer weight $\varphi_i$, determined, for example, by the user's contribution to the grid infrastructure or by the price he is willing to pay for the services he receives. We assume, without loss of generality, that the smallest task weight is equal to one.

In order to clarify the weighted max-min fair sharing, the example of Section 5.1.1 is modified as follows: Let us assume that the second and the fourth task are of twice importance compared to the first and the third task. That is, the weights are

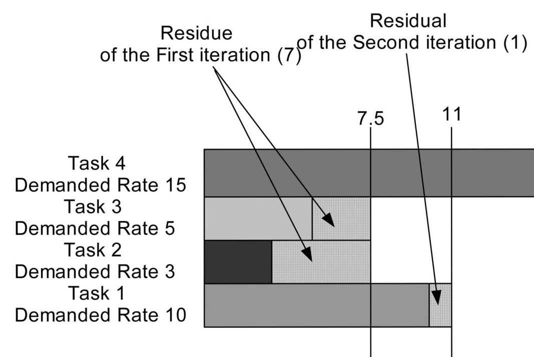



Fig. 4. A graphical conceptualization of the examples in Table 1.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

TABLE 2
An Example of the Weighted Max-Min Fair Sharing Algorithm If the Overall Processor Capacity is 30

| Demanded Rates | Weights | Max-Min Fair Sharing (First iteration) | Residual Capacity (First Iteration) | Max-Min Fair Sharing (Second iteration) |
|---|---|---|---|---|
| 10 | 1 | 5 | 0 | 7.333 |
| 3 | 2 | 3 | 7 | 3 |
| 5 | 1 | 5 | 0 | 5 |
| 15 | 2 | 10 | 0 | 14.666 |
| **Residue** | | 30-23=7 | 7 | 0 |

$1, 2, 1,$ and $2$, respectively. Since the second and the fourth task are of twice importance, we can consider that six virtual tasks demand for service (the sum of weights $1 + 2 + 1 + 2 = 6$). Therefore, the algorithm divides the total processor capacity of 30 units into six parts (the sum of the weights), that is, $30/6 = 5$ units. At the first iteration, five units are assigned to the first and third task, which have a weight of 1. Instead, 10 units $(2 * 5)$ are assigned to the second and fourth task, since their weights equal 2. However, the demanded rate for the second task is 3, which is less than the assigned rate of 10. Consequently, a residual of seven units is obtained. Instead, the demanded of the tasks 1, 3, and 4 are greater or equal to the initial assigned fair rates, yielding no residual capacity. This remaining capacity of seven units is weighted and divided to the first and fourth task whose rate are strictly less than the demanded ones so that these tasks can get an additional capacity of $7/3 = 2.333$ units (the weights of the first and fourth task equal 3). In this case, the first task gets $5 + 2.333 = 7.333$ units, whereas the fourth task gets $10 + 2.333 * 2 = 14.666$ units, and no residual capacity is observed. Thus, the algorithm terminates at the second iteration. The values obtained at each iteration are presented in Table 2.

A detailed description of the weighted max-min fair sharing algorithm is presented in Appendix B.

### 5.2 Fair Task Queue Order Estimation

As mentioned previously, a scheduling algorithm should answer two questions. First, it has to choose the order in which the tasks are considered for assignment to a processor (the *queue ordering problem*). Second, for the task that is located each time at the front of the queue, the scheduler has to decide the processor on which the task is assigned (the *processor assignment problem*). To solve the queue ordering problem in fair scheduling, we will describe shortly, in Sections 5.2.2 and 5.2.3, several ordering disciplines of different degrees of implementation complexity. Before doing so, however, we have to introduce some additional notation that will be useful in our presentation.

#### 5.2.1 Nonadjusted Fair Completion Time Estimation

We define the *nonadjusted fair completion time* $t_i$ of task $T_i$ as

$$t_i = \delta_i + w_i/r_i. \tag{8}$$

$t_i$ can be thought of as the time at which the task would be completed if it could obtain constant computation rate equal to its fair computation rate $r_i$, starting at time $\delta_i$ (recall that $\delta_i$ is the mean grid access time for task $T_i$). Note that finishing all tasks at their fair completion time is unrealistic because the grid is not really a single computer that can be accessed by user $i$ at any desired computation rate $r_i$ at a uniform delay $\delta_i$. More precisely, 1) the task is actually assigned to a specific processor $j$ and the earliest starting time on that processor is $\delta_{ij}$, 2) even if $r_i < c_j$, it may not be possible to execute the task at rate $r_i$ on that processor (we do not assume that time sharing is supported), and 3) the estimates $w_i$ of the task workloads may be inaccurate. The nonadjusted fair completion times $t_i$ are used by our algorithm as an index for determining the order in which tasks are processed by the scheduler.

#### 5.2.2 Simple Fair Task Order (SFTO)

According to the SFTO rule, the tasks are ordered in the queue in increasing order of their nonadjusted fair completion times $t_i$. In other words, the task that is first considered for assignment to a processor is the one for which it would be "fair" to finish sooner. As described earlier, the nonadjusted fair completion times are estimated from the nonadjusted computational rates $r_i$, which are in turn estimated from the tasks' demanded rates $X_i$ and the total grid processor capacity $C$. The SFTO rule is simple to implement, but it is not as fair as some of the other rules described in the following. Its performance and its fairness characteristics are rather good as shown by the simulation results presented in Section 7.

It should be mentioned that in the proposed fair scheduling algorithms, the task fair rates are approximately estimated by taking into consideration the total offered capacity of all $M$ processors. However, the task assignment exploits the properties of each individual processor resulting in a multiprocessor schema.

#### 5.2.3 Adjusted Fair Task Order (AFTO)

An issue not addressed in the definition of the nonadjusted fair completion times given in Section 5.2.1 and in the SFTO scheme presented in Section 5.2.2 is that, when tasks become inactive (because they complete execution), more capacity becomes available to be shared among the active tasks, and the fair rate of the active tasks should increase. Also, when new tasks become active (because of new arrivals), the fair rate of existing tasks should decrease. Therefore, the fair computational rate of a task is not really a constant $r_i$, as assumed so far, but it is a function of time that increases when tasks complete execution and decreases when new tasks arrive. By accounting for this time-dependent nature of the fair computational rates, the *adjusted fair completion times*, denoted by $t_i^a$, can be calculated, which can better approximate the notion of max-min fairness. In the AFTO scheme, the tasks are ordered in the queue in increasing order of their adjusted fair completion times $t_i^a$. The AFTO scheme results in schedules that are fairer than those produced by the SFTO rule; it is, however, more difficult to implement and more

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

computationally demanding than the SFTO scheme, since the adjusted fair completion times $t_i^a$ are more difficult to obtain than the nonadjusted fair completion times $t_i$. The way the adjusted fair completion times can be computed is described in Section 5.2.4. Simulation results on the computation complexity of all schemes will be presented in Section 7.

#### 5.2.4 Adjusted Fair Completion Times Estimation

To compute the adjusted fair completion times $t_i^a$, the fair rate of the active tasks at each time instant must be estimated. This can be done in two ways. In the first approach, each time an unused processor capacity is available, it is equally divided among all active tasks. In the second approach, the rates of all active tasks are recalculated using the max-min fair sharing algorithm, as described in Section 5.1, based on their respective demanded rates. The first approach is considerably less computationally intensive than the second one, since the max-min fair sharing algorithm is activated only once. The second approach, however, yields a schedule that is fairer. Regardless of the approach used, the estimated fair rate of each task is a function of time, denoted by $r_i(t)$.

Having estimated the fair rates $r_i(t)$, the fair completion time can be obtained using the following algorithm. We assume that the rates $r_i(t)$ of all tasks have been normalized so that the minimum fair task rate equals 1. We introduce a variable called the *round number*, which defines the number of rounds of service that have been completed at a given time [40]. A noninteger round number represents a partial round of service. The round number depends on the number and the rates of the active tasks at a given time. In particular, the round number increases with a rate equal to the sum of the rates of all active tasks, that is, with a slope equal to $1/\sum_i r_i(t)$. Thus, the rate with which the round number increases changes and has to be recalculated each time a new arrival or task completion takes place.

Based on the round number, we define the *finish number* $F_i(t)$ of task $T_i$ at time $t$ as

$$F_i(t) = R(\tau) + w_i/r_i(t), \tag{9}$$

where $\tau$ is the last time a change in the number of active tasks occurred (and, therefore, the last time that the round number was recalculated), and $R(\tau)$ is the round number at time $\tau$. $F_i(t)$ is recalculated each time new arrivals or task completions take place. Note that $F_i(t)$ is *not* the time that task $T_i$ will complete its execution. It is only a service tag that we will use to determine the order in which the tasks are assigned to processors. Using (9), the adjusted fair completion times $t_i^a$ can be computed as the time at which the round number reaches the estimated finish number of the respective task. Thus,

$$t_i^a: \; R(t_i^a) = F_i(t_i^a). \tag{10}$$

As mentioned earlier, the task adjusted fair completion times determine the order in which the tasks are considered for assignment to processors in the AFTO scheme: The task with the earliest adjusted fair completion time is assigned first, followed by the second earliest, and so on.

### 5.3 Fair Processor Assignment

The SFTO scheme or the AFTO scheme is used to determine the order in which the tasks are considered for assignment to processors, but it still remains to determine the particular processor where each task is assigned. A simple and efficient way to do the processor assignment is to use the ECT rule, modified so that it exploits the capacity gaps (Section 4.1). According to this rule, each task is assigned to the processor that yields the ECT. Simulation results on the performance of the SFTO and AFTO schemes when combined with the ECT rule are described in Section 7.

## 6 Max-Min Fair Scheduling (MMFS)

In this section, we present an alternative fair scheduling scheme that simultaneously obtains a fair task queuing order and a fair processor assignment.

In this algorithm, our goal is to assign a *schedulable (actual) rate* $r_i^s$ to each task so that it is as close as possible to its fair task rate $r_i$ (derived by applying the max-min fair sharing algorithm on the demanded rates $X_i$, as described in Section 5.1). The schedulable rates $r_i^s$ are smaller or equal to the task fair rates ($r_i^s \leq r_i$), and they are chosen so as not to violate the processor capacity constraints. This is expressed in the following constrained minimization problem:

$$\min E = \min \sum_{i=1}^{N} \left| r_i^s - r_i \right| \tag{11a}$$

subject to

$$\sum_{i \in P_j} r_i^s \leq c_j \qquad P_j = \{i : T_i \text{ scheduled on } j \text{ processor}\}. \tag{11b}$$

The set $P_j$ contains all tasks scheduled on processor $j$. The total deviation $E$ of the schedulable rates from the fair rates will be referred to as the *error* of the scheduler.

The minimization of (11a) subject to the constraint of (11b) can be found using the proposed algorithm described in Appendix C. The main idea of the proposed scheme is to perform an initial processor assignment and, then, appropriately rearrange the underflowed with the overflowed processors so that a better exploitation of the processor capacity is obtained. This is illustrated with the following example (see Fig. 5) where we consider two processors with capacities 20 and 25 and six tasks with fair rates 2, 5, 5, 6, 9, and 10 that have to be scheduled. Initially, processor 1 is the overflow (the processor capacity is 20 and the sum of the rates of the assigned tasks is 21), whereas processor 2 is the underflow (the processor capacity is 25 and the sum of the rates of the assigned tasks is 16). By rearranging the task of rate 2 initially assigned to processor 2 with the task of rate 10 initially assigned to processor 1, both processors turn to the underflow state, resulting in a reduction in the error. The example of Fig. 5 illustrates one iteration of the algorithm. However, in full algorithm implementation, more iterations take place.

This scheme assigns tasks to the available processors so that their actual scheduled rates are as close as possible to their respective fair rates, but it does not guarantee that a *feasible* solution is found, that is, it does not guarantee that

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

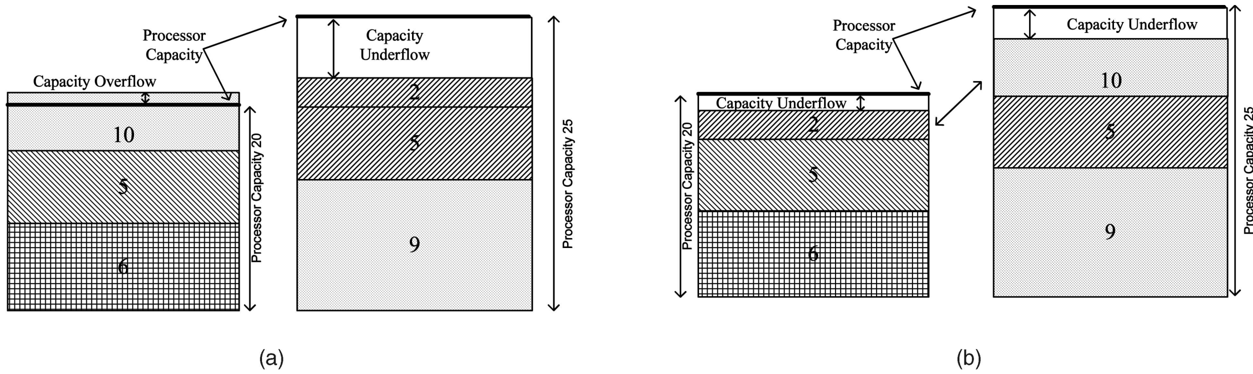



Fig. 5. The idea of the task rearrangement. (a) Initial task assignment. (b) Task rearrangement.

all tasks are assigned to the available processors without any violation of the respective deadlines.

As a result, a rate reduction is required for those tasks that are assigned to the overflow processors, so as to achieve a feasible solution. In Section 6.2, a fair rate reduction is proposed so that the feasible solution is obtained.

### 6.1 Initial Processor Arrangement

For the preceding scheme to work well, we have to start with a good initial assignment. In what follows, we present a method similar to a heuristic algorithm used in the bin-packing problem [41], which is a well-studied problem in the literature.

Initially, the algorithm sorts the tasks with respect to their fair rates in a descending order. To obtain an initial assignment, the task of the largest fair rate is first assigned to a processor, followed by the task of the second largest fair rate, and so on. The algorithm assigns, if possible, the task to a processor of adequate available capacity. If more than one processor of adequate available capacity exist, we choose the one that would leave the smallest residual capacity after the task assignment is made. In case a selected task cannot be feasibly scheduled on any processor, the task is assigned to the processor of minimal overflow. This process is terminated when all tasks have been scheduled on the available processors.

### 6.2 Fair Sharing of the Overflow Capacity and Task Queuing Order

In the previous section, we have described an algorithm for the assignment of tasks to processors so that the task schedulable rates are as close as possible to their respective fair rates. The solution obtained, however, is not necessarily feasible, since some processors may be the overflow processors. For this reason, the schedulable task rates $r_i^s$ of the overflow processors are reduced in a fair way in order to obtain a feasible solution. This is performed by the use of the max-min fair sharing algorithm as described in Section 4.

After finding the schedulable task rates (the solution also gives the processor which task each is assigned to), the tasks are scheduled for execution in an ascending order of their fair completion times on the processor to which they have been assigned. That is, the task with the ECT is first scheduled for execution, followed by the second earliest task, and so on.

## 7 Experimental Results

The proposed scheduling algorithms have been designed as part of the scheduler module of a Grid toolkit being implemented as part of the EU funded GRIDLAB (A Grid Application Toolkit and Testbed) project. A brief description of the GRIDLAB architecture is described in Section 8.1. In Section 8.2, several criteria are proposed for measuring the performance of the presented Grid scheduling algorithms, whereas in Section 8.3, simulation results and comparisons with traditional scheduling policies are given.

### 7.1 Scheduler Architecture

Fig. 6 depicts the scheduler architecture adopted in the GRIDLAB infrastructure, the main modules of which are the following:

- ***Queuing System***. Each time a task is submitted for execution, its characteristics (for example, the task deadline) are stored in a database, which is the core of the *Queuing System* module.
- ***Queuing Order***. This unit addresses the task-queue-ordering problem, that is, it determines the order in which the tasks are considered for assignment to the available resources. The queuing order unit communicates with the queuing system module, where the tasks requesting service along with their respective

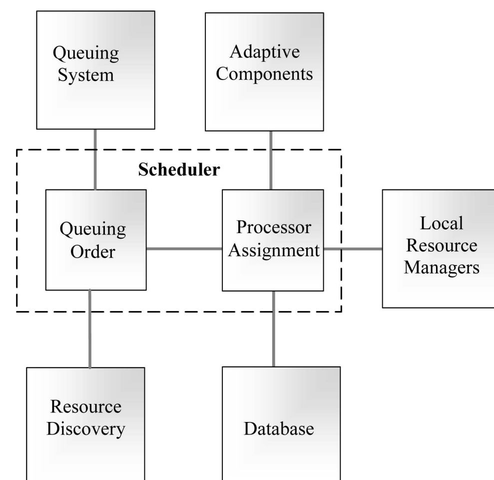



Fig. 6. The architecture of the scheduler architecture adopted in the GRIDLAB infrastructure.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

characteristics are stored and with the *Resource Discovery* module, which determines the available resources of the infrastructure.

- ***Processor Assignment***. The information collected by the *Queuing Order* unit is then passed to the *Processor Assignment* unit, which determines the processor on which each task is assigned.
- ***Adaptive Components***. This module is responsible for 1) predicting the workload of the tasks requesting service and 2) estimating the task-ready times $\delta_i$ through the respective communication delays $d_{i,j}$ and the processor release times $\gamma_j$ of the tasks already allocated to processor $j$. The adaptive component module provides the information required by the scheduler to perform the task queue ordering and the processor assignment functions. A method for predicting the task workload in the particular case of 3D rendering applications is described in [37].
- ***Resource Discovery***. This module determines the available resources of the Grid.
- ***Local Resource Manager***. This module is responsible for implementing the task execution locally as instructed by the scheduler.

In the following sections, we will compare the performance of the fair scheduling algorithms proposed in Sections 6 and 7 with that of the FCFS and the EDF policies.

### 7.2 Objective Evaluation

One criterion that we will use for measuring the performance of a scheduling algorithm is the relative error between the demanded task rates and the actual schedulable rates defined as

$$E_1 = \sum_i \frac{\left|X_i - X_i^c\right|}{X_i}, \tag{12}$$

where $X_i$ is the demanded rate, and $X_i^c$ is the actual rate allocated to the $i\text{th}$ task. Low values of error $E_1$ indicate that most of the tasks are served at rates close to their respective demanded rates.

In the FCFS and EDF algorithms, the tasks are either executed at their demanded rates $X_i$ or they are rejected. Therefore, for the FCFS and EDF schemes, the actual task rates are equal to $X_i^c = \{X_i, 0\}$ (depending on whether the task is assigned for execution or not). In contrast, in the fair scheduling schemes we proposed, all tasks are executed, possibly at a rate smaller than their demanded rate. Execution of a task with a rate smaller than its demanded rate means that the task deadline is violated.

Another criterion we will use for comparing the performance of the scheduling schemes is the ratio

$$E_2 = \frac{\sum_i X_i^c}{C}. \tag{13}$$

$E_2$ expresses the efficiency of the scheduling algorithm in allocating the available processor capacity; the greater the value of $E_2$, the better is the scheduling efficiency. When $\sum_i X_i > C$, an ideal scheduler would use the total offered processor capacity, and $E_2$ would equal 1. When $\sum_i X_i < C$, an ideal scheduler would serve all tasks with rates equal to the demanded ones. In practice, however, due to task and processor constraints (tasks are nonpreemptable, time sharing is not allowed, and so on), the ideal case cannot be achieved.

A third criterion we will use for evaluating scheduling efficiency is the average relative deviation of the demanded task deadlines to the actual task completion time:

$$E_3 = \frac{1}{N}\sum_i \frac{\left|D_i - \max(D_i^c, D_i)\right|}{D_i}, \tag{14}$$

where $D_i$ is the requested deadline and $D_i^c$ is the actual completion time of the $i\text{th}$ task. Tasks whose actual completion times are smaller than their respective deadlines do not contribute to $E_3$.

As already mentioned, the FCFS and EDF algorithms do not permit any violations of the task deadlines and they may reject tasks, in which case, the error $E_3$ becomes equal to infinity. To overcome this difficulty, we evaluate the performance of these schemes assuming that tasks whose deadline is violated are put in a *waiting list* and reapply for execution after the completion of the last feasibly assigned task.

### 7.3 Simulation Results

In this section, we simulate the proposed scheduling schemes (SFTO, AFTO, and MMFS) against 1) a large set of tasks of varying size and workload variance, 2) a large and varying number of processors, and 3) processor asymmetries (for example, groups of processors of different capacities). The statistical significance of the results is obtained by averaging the scheduling performance over 70 different runs. This statistical significance is plotting for confidence intervals of 66 percent and 95 percent, respectively. Furthermore, in the simulation results, we evaluate the effect of task deadline deviation on the performance of the proposed algorithms. Validation of the simulations under real conditions is also conducted in Section 7.3.6 for tasks generated by 3D image-rendering applications. In all experiments, the arrival model, described in Section 3.1, is adopted.

#### *7.3.1 The Effect of Task Size*

Fig. 7 presents the simulation results obtained for the SFTO, AFTO, and MMFS scheduling policies for the errors $E_1$, $E_2$, and $E_3$ against the normalized load $\rho$ (see (5)). For comparison purposes, we also depict in Fig. 7 the results obtained for the FCFS and EDF schemes. The simulations were performed assuming a Grid architecture of 500 processors with almost the same capacity (*symmetric processor case)*. In particular, we assume that the capacity of all the 500 processors follows a normal distribution with a standard deviation of 1 percent of the respective mean capacity value. In addition, we consider that 2,500 tasks arrive to the Grid within a time interval of $\Xi$ (see Section 3.1). The experiment is repeated for 20 time intervals $\Xi$, that is, for $2,500 * 20 = 50,000$ total tasks. In this experiment, we assume that all tasks present almost similar deadlines with a normal distribution of 1 percent standard deviation of the respective mean deadline value. The mean value of tasks' workload (task size) is varying, in this experiment, so that the normalized load $\rho$ takes values from 0.25 to 2.5, whereas the respective standard deviation equals 10 percent the mean value.

Under all criteria, we observe that the MMFS scheduling policy yields the highest efficiency, whereas the AFTO comes second. On the contrary, the worst performance is

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

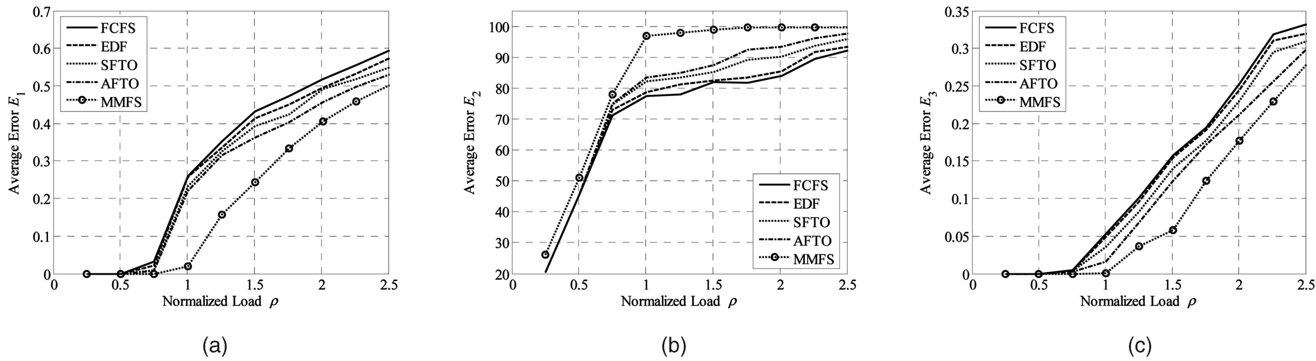



Fig. 7. The errors $E_1$, $E_2$, and $E_3$ versus the normalized load $p$ for the FCFS, the EDF, the SFTO, the AFTO, and the MMFS policies.

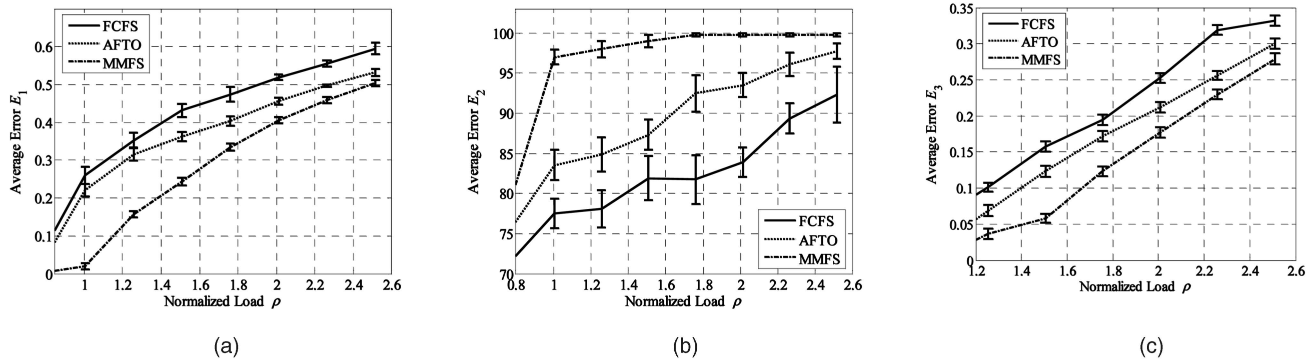



Fig. 8. The errors $E_1$, $E_2$, and $E_3$ versus the normalized load $p$ for a confidence interval of 66 percent in case of the FCFS, the AFTO, and the MMFS policies.

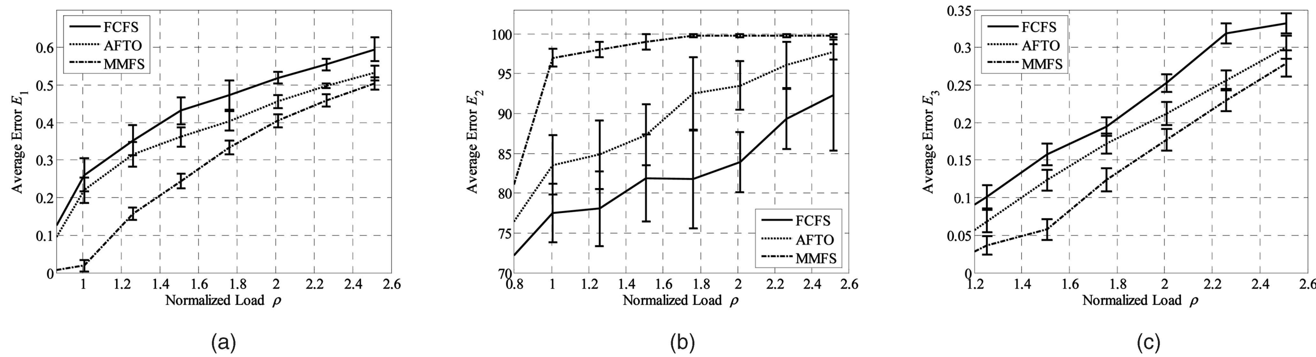



Fig. 9. The errors $E_1$, $E_2$, and $E_3$ versus the normalized load $p$ for a confidence interval of 95 percent in case of the FCFS, the AFTO, and the MMFS policies.

obtained for the FCFS policy. For light load ($\rho < 0.6$), all algorithms efficiently schedule the tasks, but as the load $\rho$ increases (that is, the tasks' size), the MMFS policy outperforms the other schemes. As is observed in error $E_2$, the MMFS performance is close to an ideal scheduler even for heavy load $\rho > 1.5$ (congestion).

Figs. 8 and 9 illustrate the statistical significance of the results of Fig. 7 by plotting the errors $E_1$, $E_2$, and $E_3$ versus $\rho$ (that is, varying of the tasks' size ) for a confidence interval of 66 percent and 95 percent, respectively. As is observed, the results slightly vary around the average value, verifying their statistical robustness. In particular, for a confidence interval of 66 percent, the worst performance of a scheduling policy (for example, MMFS) outperforms the best results of the other scheduling policies. The same conclusions are drawn for almost all cases even if a confidence interval of 95 percent is selected as shown in Fig. 9.

The effect of the tasks' deadline deviation on the error $E_1$ for varying tasks' size is shown in Fig. 10. In this figure, the performance of the SFTO, AFTO, and MMFS algorithms are compared using low and high variation in the tasks' deadlines. In particular, Fig. 10 compares the results obtained from Fig. 7a with the results obtained assuming that the deadlines follow a normal probability density function (pdf) of high standard deviation (in this experiment, as standard deviation, we select the mean value of the task deadlines). As is observed, for all algorithms except for MMFS, an improvement in the error $E_1$ is noticed. However, the MMFS scheduling efficiency deteriorates as the tasks' deadline variation increases meaning that the algorithm improvement compared to the other

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

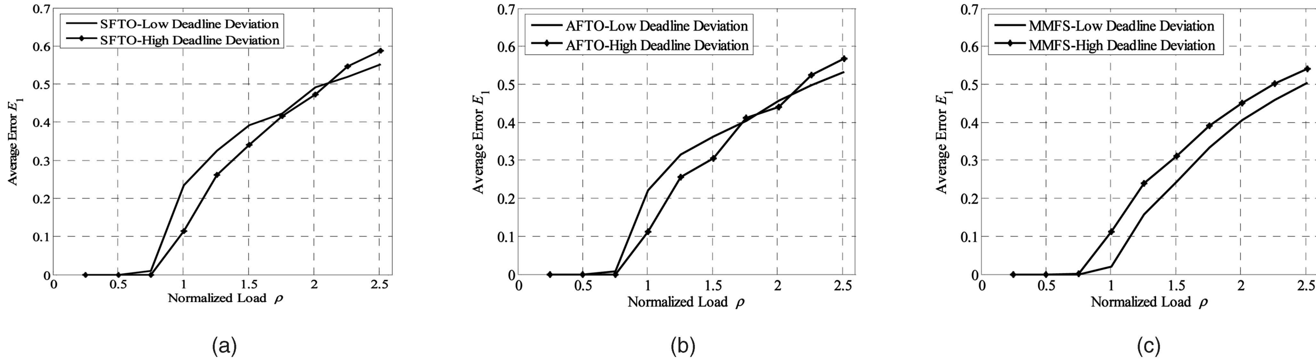



Fig. 10. The error $E_1$ versus the load $\rho$ in case of low and high deadline deviation for (a) the SFTO, (b) the AFTO, and (c) the MMFS policy.

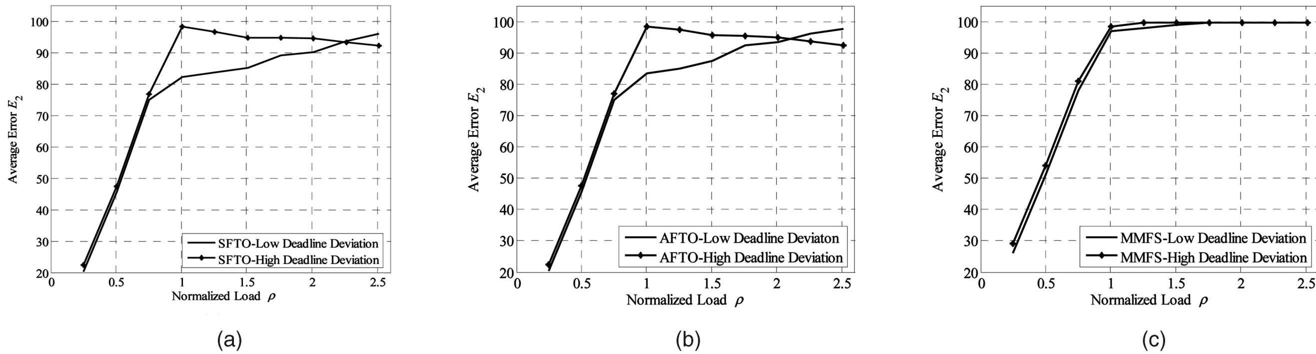



Fig. 11. The error $E_2$ versus the load $\rho$ in case of low and high deadline deviation for (a) the SFTO, (b) the AFTO, and (c) the MMFS policy.

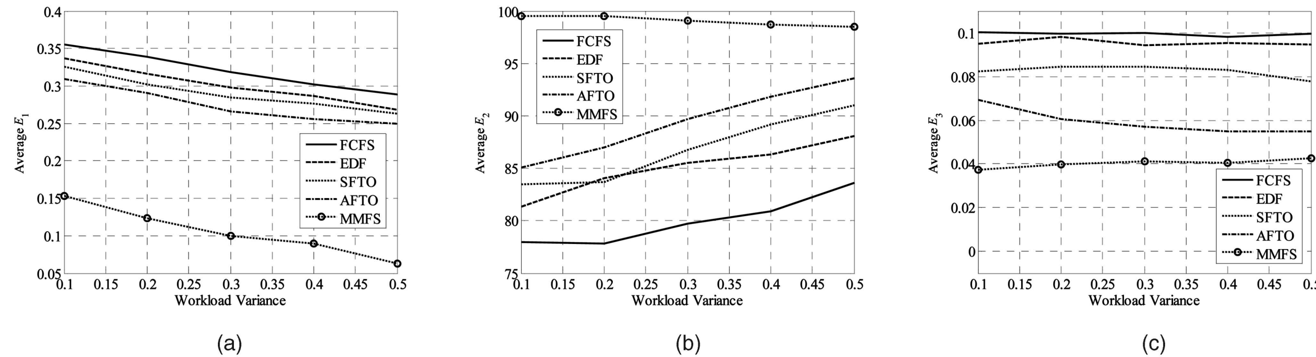



Fig. 12. The errors $E_1$, $E_2$, and $E_3$ versus the variance of the task workload for the FCFS, EDF, SFTO, AFTO, and the MMFS scheduling policies.

techniques also decreases. This is due to the fact the MMFS policy reallocates the tasks in the processors without taking into consideration their deadlines. Similar conclusions are drawn using the criterion $E_2$ (see Fig. 11).

#### 7.3.2 The Effect of Workload Variance

The effect of the task workload variance on scheduling performance is illustrated in Fig. 12. In this figure, again, a symmetric case of 500 processors is considered as in Fig. 7 along with tasks of similar deadlines and $\rho = 1.25$. The arrival model of Section 3.1 is adopted. In this experiment, the number of tasks equals 2,500 within the time interval $\Xi$. The experiment is repeated for 20 time intervals $\Xi$, that is, for $2,500 * 20 = 50,000$ total tasks. We recall that, for each experiment, 70 runs have been conducted as in the previous case, and the average value over all runs is depicted in Fig. 12. The MMFS scheme performs better than the other scheduling methods, for all values of workload variances and errors $E_1$, $E_2$, and $E_3$. Note that as the workload variance increases, the performance of all the schemes improves (except for that of MMFS, which remains almost the same since it is close to the ideal case).

The variation of the values obtained across all the 70 runs is shown in Fig. 13 for a confidence interval of 66 percent. As it is noticed, in all cases, the worst performance of the MMFS and the AFTO outperforms the best performance of the FCFS under all three criteria.

#### 7.3.3 The Effect of the Number of Processors

Fig. 14 illustrates the performance of the five examined scheduling policies with respect to the number of processors. The experiments have been conducted for a number of processors ranging in $[50, \ldots, 1,000]$. In all cases, a symmetric processor capacity is considered for a load $\rho = 1.5$. It is also assumed that the workload variance equals 10 percent of the respective mean value. As the

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

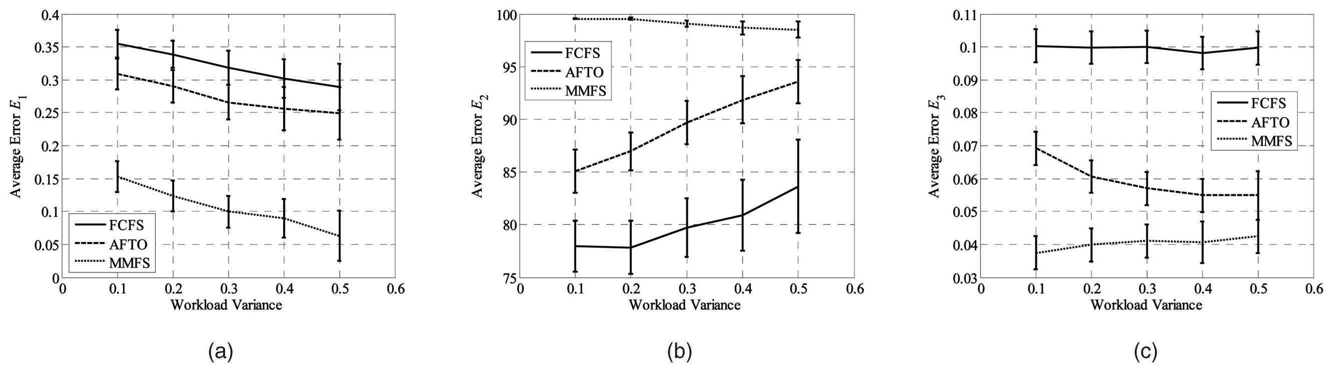



Fig. 13. The errors $E_1$, $E_2$, and $E_3$ versus the workload variance for a confidence interval of 66 percent in case of the FCFS, the AFTO, and the MMFS scheduling policies.

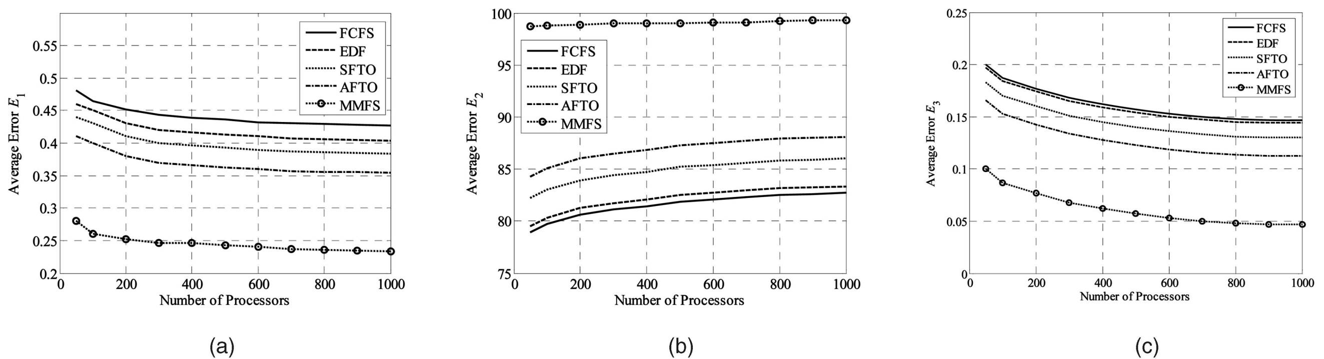



Fig. 14. The errors $E_1$, $E_2$, and $E_3$ versus the number of processors for $\rho = 1.5$ and workload variance 0.1.

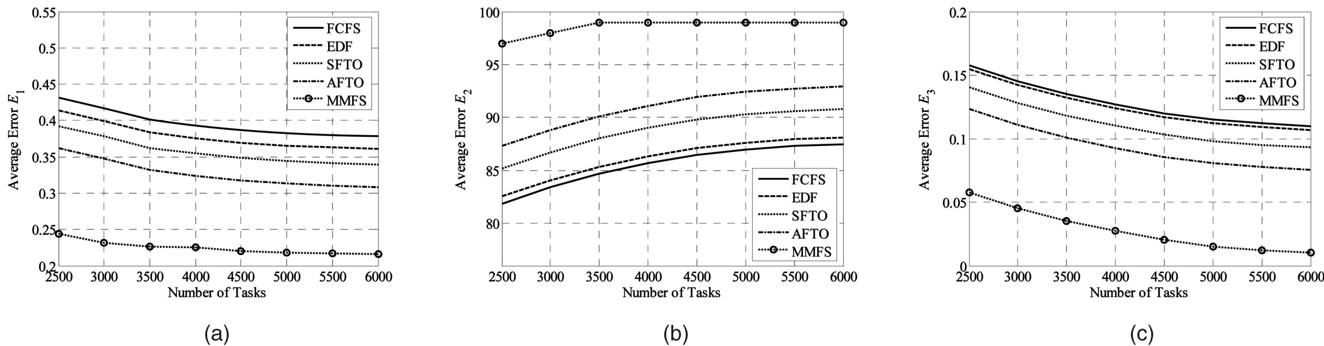



Fig. 15. The errors $E_1$, $E_2$, and $E_3$ versus the number of tasks for $\rho = 1.5$ and workload variance 0.1.

number of processors increases, the number of tasks also increases to retain a constant load to the Grid infrastructure. As is observed, the MMFS scheduling policy outperforms the other ones, with the AFTO scheme being the second best. In addition, as the number of processor increases, a slight improvement in scheduling efficiency is accomplished, with a decreasing ratio, however. For the error $E_2$, the performance of the MMFS approach is almost near to the ideal one independent from the number of the processors comprise the Grid infrastructure.

#### 7.3.4 The Effect of the Number of Tasks

Fig. 15 presents the influence of the number of tasks that arrive within the time interval $\Xi$. In this experiment, the load $\rho$ equals 1.5 and 500 processors of similar capacity are adopted. As is observed, the MMFS scheduling policy yields the best results compared to the other examined approaches. It is also observed that as the number of tasks increases, the scheduling efficiency also increases, but with a decreasing rate. As the number of tasks increases, the task size decrease to retain a constant load $\rho$. This indicates that, for a large number of small tasks, the scheduling algorithms better exploit the available processor capacity of the Grid infrastructure.

#### 7.3.5 The Effect of Processor Capacity Variation

In this section, we evaluate the effect of the processor asymmetries (variation of the processor capacity) on the scheduling performance. In all experiments, 70 runs have been conducted, and the average value over all runs is chosen. In addition, 500 processors are considered of varying capacity. Furthermore, we consider that 2,500 tasks arrive to the Grid within a time interval of $\Xi$ and that each experiment is repeated until the time reaches $20 * \Xi$, that is, for $2,500 * 20 = 50,000$ total tasks.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

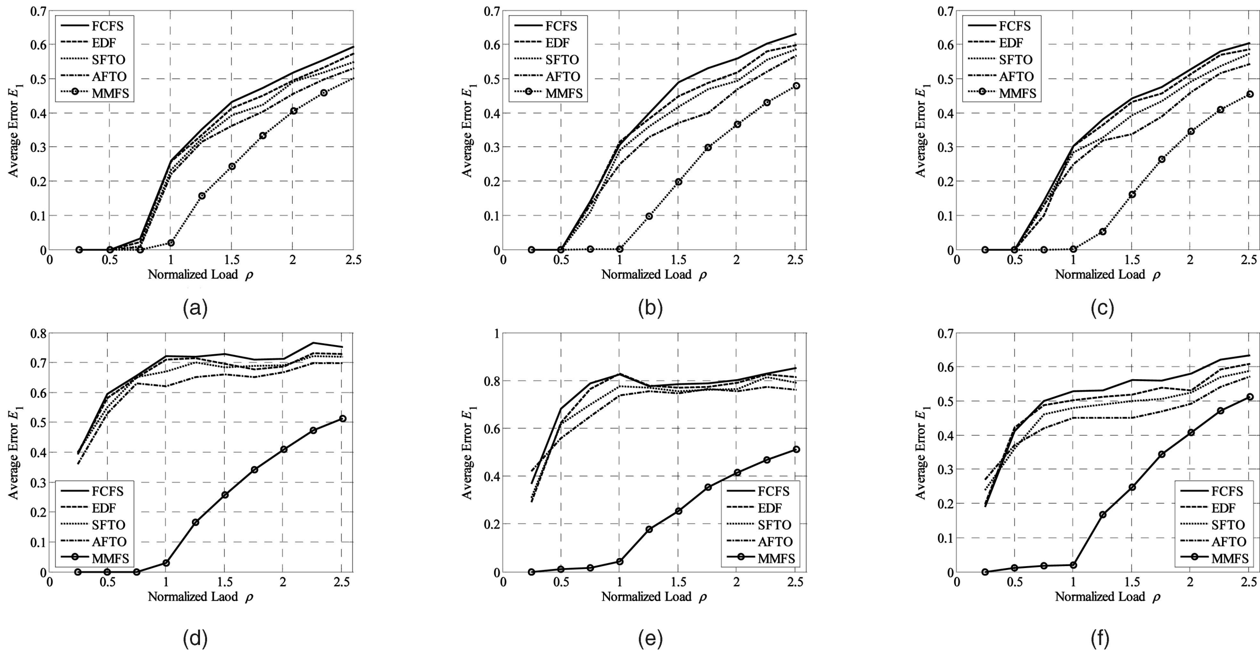



Fig. 16. The effect of processor capacity variation on the error $E_1$ (a) symmetric case, (b) asymmetric Gaussian case, (c) uniform case, (d) cluster symmetric case, (e) cluster asymmetric case biased of low processor capacity, and (f) cluster asymmetric case biased of high capacity.

In particular, Fig. 16 illustrates the performance for the five examined scheduling schemes with respect to the normalized $\rho$ load under different scenarios of processor capacity for the error $E_1$. In all experiments, the *same mean processor* capacity is considered. Fig. 16a shows the results of processors of similar capacities (as in Fig. 7a), whereas Figs. 16b, 16c, 16d, and 16f show the results obtained using asymmetric processor capacities. In particular, Fig. 16b considers that the capacity of the processors follows a normal pdf of high standard deviation (in this case, the half of the respective mean value), called *Asymmetric Gaussian case*. Fig. 16c assumes a uniform distribution of the processor capacity in the interval of $[\mathrm{a}2 * \mu + \mathrm{a}]$, where $a$ is a small value of processor capacity, and $\mu$ is the mean processor value (this is called the *Asymmetric Uniform case*). The effect of different groups of processors, with processors in each group having the same capabilities, is presented in Figs. 16d, 16e, and 16f. In the scenario of Fig. 16d, the processors are equally divided into two groups, one of high capacity and one of low capacity—*Cluster Symmetric case*. Fig. 16e presents the scenario where the majority of the processors (80 percent) are of low capacity, whereas the remaining are of high one—*Cluster Biased Low case*. In contrast, Fig. 16f shows the opposite scenario, where the majority of the processors (of 80 percent) are of high capacity—*Cluster Biased High case*.

As observed, in all cases, the MMFS outperforms the other examined scheduling schemes. In addition, we notice that high deviation of the processors' capacities deteriorates the scheduling performance for the FCFS, EDF, SFTO, and AFTO policies. Instead, the MMFS scheme remains robust. This is due to the fact that the task reallocation adopted in the MMFS method optimizes the task assignment to the processors, which is not achieved by the other approaches. This is more evident in case of many low capacity processors, since the few processors of high capacity cannot compensate the infeasible scheduling of tasks assigned to low-capacity processors.

Comparisons of the error $E_1$ for the six different examined cases of the processors' variations are presented in Fig. 17 for the five different scheduling policies. In this figure, the aforementioned conclusions are more evident. More specifically, the best performance for FCFS, EDF, SFTO, and AFTO is accomplished for the symmetric case and the worst for the case wherein many processors of low capacity exist. It is also noticeable that the differences are not so important in case that the processors' capacity is allocated over all possible values (see the normal and uniform distribution). On the contrary, the existence of processor groups with extreme capacity values yields significance deterioration of the results (better performance in a case biased of high processor capacity is accomplished).

#### 7.3.6 Validation in Real Experiments

Validation of the aforementioned simulations in real experiments is presented in Fig. 18. The experiment has been conducted using the GRIA and GRIDLAB architecture as described in Section 7.1. In this case, 50 processors consist of the cluster of the Grid infrastructure. Tasks are generated from 3D rendering algorithms in computer graphics. The validation is performed assuming an arrival of 250 tasks within the $\Xi$ time intervals. All tasks are assumed to have similar deadlines. Similar conclusions are observed in this case, which validates the simulations conducted in the previous sections. More specifically, the MMFS yields the best performance with the AFTO scheduling policy comes the second.

#### 7.3.7 Computational Complexity

The computational complexity of the EDF, SFTO, AFTO, and MMFS scheduling schemes is presented in Table 3 for different values of tasks requested service and different number of processors. The worse expressions for the algorithm complexity over all the 70 runs are taken into account. The computational complexity has been normalized

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

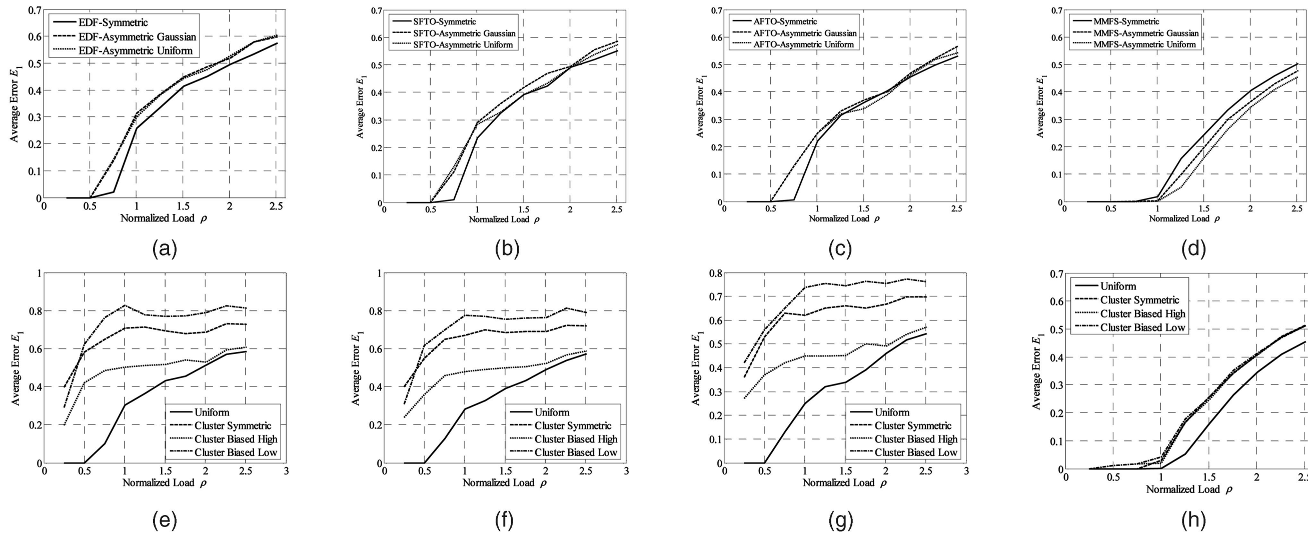



Fig. 17. Comparison of the processor capacity variation for the error $E_1$. (a), (b), (c), and (d) The symmetric case, the asymmetric Gaussian case, and the uniform case for the EDF, SFTO, AFTO, and MMFS, respectively. (e), (f), (g), and (h) The cluster symmetric case, the cluster asymmetric case biased of high processor capacity, and the cluster asymmetric case biased of low capacity for the EDF, SFTO, AFTO, and MMFS, respectively.

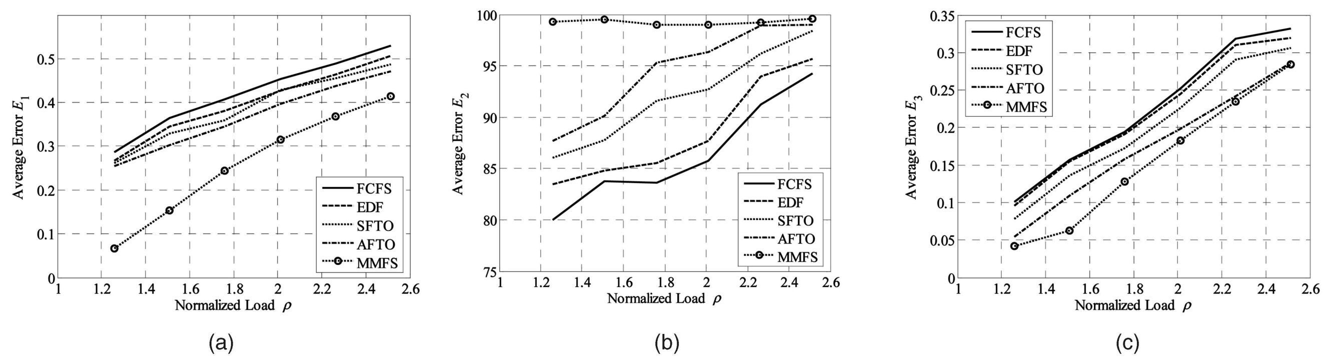



Fig. 18. The scheduling performance on a real experiment of tasks generated by 3D rendering computer graphics algorithms with (a) the error $E_1$, (b) the error $E_2$, and (c) the error $E_3$.

with respect to the cost of the EDF scheme when the number of submitted tasks equals 2,500 and the number of processors equals 100. As expected, the computational complexity increases with respect to the number of tasks and with respect to the number of processors. The MMFS policy is sensitive to the number of processors, though in our experiments, it presents a low computational complexity for a relatively small number of processors. However, it should be mentioned that the computational complexity of the MMFS can be retained very small if a constant number of iterations is adopted at the expense of the performance efficiency. Instead, the AFTO policy is sensitive to the number of the submitted tasks, since it requires reestimation of the fair rates upon task arrivals or departures. The complexity of the other scheduling policies is almost linearly affected with respect to the number of tasks and processors.

# 8 Conclusions

In this paper, we proposed three new scheduling algorithms for the Grid environment that could be used to implement scheduling in a fair way. In the SFTO, the tasks are ordered in the queue in an increasing order with respect to their nonadjusted fair completion times. The nonadjusted fair completion times are obtained by the nonadjusted fair computational rates of the tasks by applying a max-min fair sharing algorithm. An improved version of the SFTO scheme is the AFTO scheduling policy, where the fair rates are dynamically adjusted each time tasks become inactive (for example, they complete execution) or active (for example, new arrivals) to better exploit the offered multiprocessor capacities. In the AFTO scheme, the fair rates of the tasks are not constant, as is assumed in the SFTO scheme, but they increase when tasks complete execution and decrease when new tasks arrive. In both scheduling methods, the processor at which the tasks are assigned for execution is found based on the ECT policy modified so that processor capacity gaps are taken into account. Finally, in the third fair Grid scheduling scheme that we presented, the max-min fair scheduling (MMFS) scheme, the selection of a fair task queuing order and the selection of a processor assignment are simultaneously addressed. In particular, the algorithm allocates the tasks to the available processors so that the actual task rate after the scheduling is much as close to the fair ones as obtained by the max-min fair sharing scheme. All the three proposed scheduling algorithms are oriented to a multiprocessor computing system. The three proposed algorithms can be integrated in existing Grid computing systems to improve the task allocation performance.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

TABLE 3
The Normalized Computational Complexity of the EDF, SFTO, AFTO, and MMFS with Respect to the Number of Tasks and the Number of Processors (Four Instances)

| | Number of Processors 100 | | | | |
|---|---|---|---|---|---|
| Number of Tasks | 2500 | 3500 | 4500 | 5500 | 6500 |
| EDF | 1.00 | 1.61 | 2.12 | 2.72 | 3.05 |
| SFTO | 1.15 | 1.81 | 2.46 | 3.21 | 3.70 |
| AFTO | 1.63 | 2.98 | 3.98 | 4.43 | 5.32 |
| MMFS | 0.75 | 0.92 | 1.25 | 1.42 | 1.61 |
| | Number of Processors 300 | | | | |
| Number of Tasks | 2500 | 3500 | 4500 | 5500 | 6500 |
| EDF | 3.02 | 4.91 | 6.41 | 8.19 | 9.17 |
| SFTO | 3.47 | 5.52 | 7.40 | 9.73 | 11.13 |
| AFTO | 4.91 | 9.02 | 11.99 | 13.41 | 16.01 |
| MMFS | 4.17 | 5.14 | 6.75 | 7.56 | 8.88 |
| | Number of Processors 500 | | | | |
| Number of Tasks | 2500 | 3500 | 4500 | 5500 | 6500 |
| EDF | 5.05 | 8.14 | 10.71 | 13.74 | 15.40 |
| SFTO | 5.88 | 9.10 | 12.44 | 16.19 | 18.67 |
| AFTO | 8.34 | 15.01 | 20.09 | 22.25 | 26.85 |
| MMFS | 7.72 | 9.51 | 13.4 | 14.84 | 16.70 |
| | Number of Processors 700 | | | | |
| Number of Tasks | 2500 | 3500 | 4500 | 5500 | 6500 |
| EDF | 7.12 | 11.45 | 15.01 | 19.38 | 21.82 |
| SFTO | 8.11 | 12.92 | 17.61 | 22.95 | 26.30 |
| AFTO | 11.60 | 21.20 | 28.44 | 31.57 | 37.92 |
| MMFS | 15.32 | 18.71 | 25.43 | 28.90 | 32.98 |

*Normalization is performed for the EDF at 100 processors and 2,500 tasks.*

Experimental results and comparisons with the traditional FCFS and EDF scheduling schemes indicate that our proposed scheduling schemes are fairer and better exploit the available multiprocessor Grid resources. The simulations have been conducted by submitted thousands of tasks of varying size and workload variance to a multiprocessor computing system comprising of hundreds of processors of varying capacity. The experiments indicate that the MMFS algorithm is less sensitive to processor capacity variations instead of the SFTO and AFTO scheme. However, in all conditions, the proposed algorithms outperform the traditional ones. The AFTO policy is more effective than the SFTO one in the extent of computational complexity. The MMFS policy outperforms the SFTO and the AFTO schemes in all the simulation conditions.

## APPENDIX A

## IDEAL NONWEIGHTED MAX-MIN FAIR SHARING ALGORITHM

The nonweighted max-min fair sharing algorithm is described as follows: The demanded computation rates $X_i$, $i = 1, 2, \ldots, N$, of the tasks are sorted in ascending order, say, $X_1 < X_2 < \cdots < X_N$. Initially, we assign capacity $C/N$ to the task $T_1$ with the smallest demand $X_1$, where $C$ is the total grid computation capacity (1). If the fair share $C/N$ is more than the demanded rate $X_1$ of task $T_1$, the unused excess capacity of $C/N - X_1$ is again equally shared to the remaining tasks $N-1$ so that each of them gets an additional capacity $(C/N + (C/N - X_1))/(N-1)$. This may be larger than what task $T_2$ needs, in which case, the excess capacity is again equally shared among the remaining $N-2$ tasks, and this process continues until there is no computation capacity left to distribute or until all tasks have been assigned capacity equal to their demanded computation rates. When the process terminates, each task has been assigned no more capacity than what it needs, and, if its demand was not satisfied, no less capacity than what any other task with a greater demand has been assigned. This scheme is called nonweighted max-min fair sharing since it maximizes the minimum share of a task whose demanded computation rate is not fully satisfied.

We can mathematically describe the previous algorithm as follows. We denote by $r_i(n)$ the *nonadjusted fair computation rate* of the task $T_i$ at the $\mathrm{n^{th}}$ iteration of the algorithm. Then, $r_i(n)$ is given by

$$r_i(n) = \begin{cases} X_i & \text{if } X_i < \sum_{k=0}^{n} O(k) \\ \sum_{k=0}^{n} O(k) & \text{if } X_i \geq \sum_{k=0}^{n} O(k), \end{cases} \quad n \geq 0, \qquad \text{(A1a)}$$

where

$$O(n) = \frac{C - \sum_{i=1}^{N} r_i(n-1)}{card\{N(n)\}}, n \geq 1 \qquad \text{(A2b)}$$

with

$$O(0) = C/N. \qquad \text{(A1c)}$$

In (A1b), $N(n)$ is the set of tasks whose assigned fair rates are smaller than their demanded computation rates at the beginning of the $\mathrm{nth}$ iteration, that is,

$$N(n) = \{T_i : X_i > r_i(n-1)\} \text{ and } N(0) = N, \qquad \text{(A2)}$$

whereas the function $card(\cdot)$ returns the cardinality of a set. The process is terminated at the first iteration $n_o$ at which either $O(n_o) = 0$ or the number $card\{N(n_o)\} = 0$. The former case indicates congestion, whereas the latter indicates that the total grid computation capacity can satisfy all the demanded task rates, that is,

$$\sum_{i=1}^{N} X_i < C. \qquad \text{(A3)}$$

The nonadjusted fair computation rate $r_i$ of task $T_i$ is obtained at the end of the process as

$$r_i = r_i(n_0). \qquad \text{(A4)}$$

## APPENDIX B

## IDEAL WEIGHTED MAX-MIN FAIR SHARING ALGORITHM

In this case, we allocate computation capacity as if the number of submitted tasks is equal to the sum of the respective weights, that is, as if there were $\tilde{N} = \sum_{i=1}^{N} \varphi_i$ virtual tasks. An equal fair sharing is performed for all $\tilde{N}$ virtual tasks using the algorithm of Section 6.1.1. Equation (A1) is then modified as follows:

$$r_i(n) = \begin{cases} X_i & \text{if } X_i < \varphi_i \cdot \sum_{k=0}^{n} O(k) \\ \varphi_i \cdot \sum_{k=0}^{n} O(k) & \text{if } X_i \geq \varphi_i \cdot \sum_{k=0}^{n} O(k), \end{cases} \quad n \geq 0, \qquad \text{(B1)}$$

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

where

$$O(n) = \frac{C - \sum_{i=1}^{N} r_i(n-1)}{\tilde{N}(n)}, n \geq 1 \tag{B2}$$

and

$$O(0) = C/\tilde{N}, \text{ with } \tilde{N} = \sum_{i=1}^{N} \varphi_i. \tag{B3}$$

$\tilde{N}(n)$ is the sum of the weights of the tasks whose assigned fair rates are smaller than their demanded computation rates at the beginning of the nth iteration of the algorithm, that is,

$$\tilde{N}(n) = \sum_i \varphi_i : \text{for all } i : X_i > r_i(n-1) \text{ and } \tilde{N}(0) = \tilde{N}. \tag{B4}$$

The process is terminated at an iteration $n_o$ at which either $O(n_o) = 0$ or $card\{\tilde{N}(n_o)\} = 0$.

## Appendix C

## Max-Min Fair Scheduling

Since tasks are nonpreemptable (they cannot be split in smaller units that are executed on different processors), the sum of the rates of the tasks assigned for execution to a processor may be smaller than the processor capacity, and some processors may not be fully utilized. A processor with unused capacity will be called an *underflow* processor. In an optimal solution, tasks assigned to underflow processors have schedulable rates that are equal to their respective fair rates, $r_i^s = r_i$, and do not contribute to the error $E$ (otherwise, we could assign additional capacity to those tasks and reduce the error). Only tasks assigned to fully utilized processors may contribute to the error $E$ (but not all tasks assigned to fully utilized processors contribute to the error).

We define the overflow $O_j$ of processor $j$ as

$$O_j = \max\left\{0, \sum_{i \in P_j} r_i - c_j\right\} \tag{C1a}$$

and the underflow $U_k$ of processor $k$ as

$$U_k = \min\left\{0, \sum_{i \in P_k} r_i - c_k\right\}. \tag{C1b}$$

Processors for which $O_j > 0$ will be referred to as *overflow* processors, whereas *underflow* processors are those for which $U_k < 0$. In an optimal solution, we have

$$\sum\nolimits_{i \in P_j} r_i^s = c_j, \text{ for all } j \text{ for which } O_j > 0$$

and the error $E$ is equal to the sum of the total processor overflow

$$E = \sum_{i=1}^{N} \left| r_i - r_i^s \right| = \sum_{j \in \Gamma} O_j, \tag{C2}$$

where $\Gamma$ is the set of overflow processors (underflow processors do not contribute to the error $E$).

Therefore, the minimization problem of (11) can be rewritten as

$$\min E = \min \sum_{j \in \Gamma} O_j \tag{C3a}$$

subject to

$$\sum_{i \in P_j} r_i^s \leq c_j, P_j = \{i : T_i \text{ scheduled on } j \text{ processor}\}. \tag{C3b}$$

Equation (C3) states that scheduling the tasks with rates as close as possible to their respective fair rates (11) is equivalent to finding a solution that minimizes the overall processor overflow. However, the minimization of (C3a) subject to the constraint of (C3b) is computationally intensive (it is the similar to the bin-packing problem, which is NP-complete), since every possible task assignment to the $M$ available processors should be examined. In what follows, we propose a heuristic task-rearrangement scheme of polynomial time to obtain a good assignment of tasks to processors.

**Processor Assignment**. The proposed algorithm combines processors of capacity overflow with processors of capacity underflow to obtain a better exploitation of the overall processor capacity. More specifically, given an assignment of tasks to processors, we consider the rearrangement where a task of rate $r_l$ assigned to an overflow processor is substituted for a task of rate $r_m$ assigned to an underflow processor. After the task rearrangement, the overflow (underflow) capacity of the processors is updated as follows:

$$R_j = O_j - \varepsilon, \tag{C4a}$$

$$R_k = U_k - \varepsilon, \tag{C4b}$$

where

$$\varepsilon = r_m - r_l. \tag{C4c}$$

Equation (C4c) expresses the task rate difference between the two selected tasks, where $R_j$ and $R_k$ are the updated processor residuals. If $R_j > 0$, processor $j$ remains at the overflow state after the task rearrangement, whereas if $R_j < 0$, processor $j$ turns to the underflow state.

The tasks with rates $r_l$ and $r_m$ that are exchanged are selected so as to reduce the overall processor overflow or equivalently reduce the error $E$ given by (C2). A reduction is accomplished only if the task rate difference as expressed by (C4c) satisfies the following equation:

$$\varepsilon : O_j' + O_k' < O_j, \tag{C5}$$

where $O_j' = \max(0, R_j)$ and similarly $O_k' = \max(0, R_k)$.

The explanation of (C5) is stated as follows: Initially, processor $j$ is the overflow, and processor $k$ is the underflow, so that only processor $j$ contributes to the error $E$ (see (C3)). After the task rearrangement, the total contribution of the processors $j$ and $k$ to the error $E$ should be less than their initial contribution for the rearrangement to yield an error reduction. Such rearrangements between the overflow and underflow processors are repeated until no task rearrangement that satisfies (C5) can be found.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

## REFERENCES


[1] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Int'l J. Supercomputer Applications,* vol. 15, no. 3, 2001.

[2] W. Leinberger and V. Kumar, "Information Power Grid: The New Frontier in Parallel Computing," *IEEE Concurrency,* vol. 7, no. 4, pp. 75-84, Oct.-Dec. 1999.

[3] "Scheduling Working Group of the Grid Forum," Document: 10.5, Sept. 2001.

[4] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Int'l J. Supercomputer Applications,* vol. 11, no. 2, pp. 115-128, 1997.

[5] J. Basney, M. Livny, and T. Tannenbaum, "High Throughput Computing with Condor," *High Performance Computer Unit (HPCU) News,* vol. 1, no. 2, June 1997.

[6] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," *Grid Computing: Making the Global Infrastructure a Reality,* F. Berman, A.J.G. Hey, and G. Fox, eds., John Wiley & Sons, 2003.

[7] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *J. Cluster Computing,* vol. 5, pp. 237-246, 2002.

[8] A.S. Grimshaw, M.A. Humphrey, and A. Natrajan, *A Philosophical and Technical Comparison of Legion and Globus.* Corp. Riverton, 2004.

[9] D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '00),* 2000.

[10] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R.R. Sutherst, and N. White, "Nimrod Computational Workbench: A Case Study in Desktop Metacomputing," *Proc. Australian Computer Science Conf. (ACSC '97),* Feb. 1997.

[11] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," *Proc. Fourth Int'l Conf. High Performance Computing in Asia-Pacific Region,* 2000.

[12] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software Support for High-Level Grid Application Development," *Int'l J. High Performance Computing Applications,* vol. 15, no. 4, pp. 327-344, Winter, 2001.

[13] H. Dail, H. Casanova, and F. Berman, "A Decoupled Scheduling Approach for the GrADS Environment," *Proc. Conf. Supercomputing (SC '02),* Nov. 2002.

[14] R. Wolski, J.S. Plank, J. Brevik, and T. Bryan, "G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '01),* Apr. 2001.

[15] S.M. Jackson, "Allocation Management with QBank," white paper, technical report in Pacific Northwest Nat'l Laboratories, 2000.

[16] T. Hacker and W. Thigpen, "Distributed Accounting on the Grid," *Grid Forum Working Draft,* 2007.

[17] M.S. Fineberg and O. Serlin, "Multiprogramming for Hybrid Computation," *Proc. Int'l Federation for Information Processing Societies (IFIPS) Fall Joint Computer Conf.,* 1967.

[18] J.A. Stankovic, et al. "Implications of Classical Scheduling Results for Real Time Systems," *Computer,* pp. 16-25, June 1995.

[19] M.L. Dertouzos and A.K.-L. Mok, "Multiprocessor On-Line Scheduling for Hard Real Time Tasks," *IEEE Trans. Software Eng.,* pp. 1497-1506, Dec. 1989.

[20] G. Manimaran, C.S.R. Murthy, M. Vijay, and K. Ramamritham, "New Algorithms for Resource Reclaiming from Precedence Constrained Tasks in Multiprocessor Real-Time Systems," *J. Parallel and Distributed Computing,* vol. 44, no. 2, pp. 123-132, Aug. 1997.

[21] K. Ramamritham, J.A. Stankovic, and P.-F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 2, pp. 184-194, Apr. 1990.

[22] W. Zhao, K. Ramamritham, and J.A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real Time Systems," *IEEE Trans. Software Eng.,* vol. 12, no. 3, pp. 360-369, May 1990.

[23] J.Y-T. Leung and M.L. Merrill, "A Note on Preemptive, Scheduling of Periodic, Real-Time Tasks," *Information Processing Letters,* pp. 115-118, Nov. 1980.

[24] X. Deng, N. Gu, T. Brecht, and K.-C. Lu, "Preemptive Scheduling of Parallel Jobs on Multiprocessors," *SIAM J. Computing,* vol. 30, no. 1, pp. 145-160, 2000.

[25] G. Manimaran and C.S.R. Murthy, "An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 3, pp. 312-319, Mar. 1998.

[26] L.E. Jackson and G.N. Rouskas, "Deterministic Preemptive Scheduling of Real Time Tasks," *Computer,* vol. 35, no. 5, pp. 72-79, May 2002.

[27] W. Zhang, B. Fang, H. He, H. Zhang, and M. Hu, "Multisite Resource Selection and Scheduling Algorithm on Computational Grid," *Proc. 18th Parallel and Distributed Processing Symp.,* pp. 105-115, 2004.

[28] S. Zhuk, A. Chernykh, A. Avetisyan, S. Gaissaryan, D. Grushin, N. Kuzjurin, A. Pospelov, and A. Shokurov, "Comparison of Scheduling Heuristics for Grid Resource Broker," *Proc. IEEE Fifth Mexican Int'l Conf. Computer Science,* pp. 388-392, 2004.

[29] D.P. Spooner, S.A. Jarvis, J. Cao, S. Saini, and G.R. Nudd, "Local Grid Scheduling Techniques Using Performance Prediction," *IEE Proc. Computers and Digital Techniques,* vol. 150, no. 2, pp. 87-96, Mar. 2003.

[30] S. Kim and J.B. Weissman, "A Genetic Algorithm Based Approach for Scheduling Decomposable Data Grid Applications," *Proc. Int'l Conf. Parallel Processing (ICPP '04),* pp. 406-413, 2004.

[31] R. Jain, *A Survey of Scheduling Methods.* Nokia Research Center, Sept. 1997.

[32] M. Hawa, "Stochastic Evaluation of Fair Scheduling with Applications to Quality-of-Service in Broadband Wireless Access Networks," PhD dissertation, Univ. of Kansas, Aug. 2003.

[33] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and K. Li, "Experimental Performance Evaluation of Job Scheduling and Processor Allocation Algorithms for Grid Computing on Metacomputers," *Proc. IEEE 18th Int'l Parallel and Distributed Processing Symp. (IPDPS '04),* pp. 170-177, 2004.

[34] A.K. Parekh and R.G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Trans. Networking,* vol. 1, no. 3, pp. 344-357, 1993.

[35] A. Demers, S. Keshav, and S. Shenker, "Design and Analysis of a Fair Queuing Algorithm," *Proc. ACM SIGCOMM '89,* Sept. 1989.

[36] D. Bertsekas and R. Gallager, *Data Networks,* second ed., section starting on p. 524. Prentice Hall, 1992.

[37] N. Doulamis, A. Doulamis, A. Panagakis, K. Dolkas, T. Varvarigou, and E. Varvarigos, "A Combined Fuzzy-Neural Network Model for Non-Linear Prediction of 3D Rendering Workload in Grid Computing," *IEEE Trans. Systems Man and Cybernetics, Part-B,* vol. 34, no. 2, pp. 1235-1247, Apr. 2004.

[38] J. Turner, "Terabit Burst Switching," *J. High Speed Networks,* vol. 8, no. 1, pp. 3-16, 1999.

[39] Y. Xiong, M. Vandenhoute, and H.C. Cankaya, "Control Architecture in Optical Burst-Switched WDM Networks," *IEEE J. Selected Areas in Comm.,* vol. 18, pp. 1838-1851, 2000.

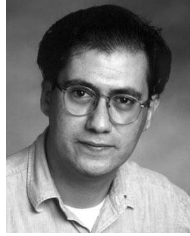
[40] S. Keshav, *An Engineering Approach to Computer Networking.* Addison-Wesley, 1997.

[41] D.S. Johnson, "Fast Algorithms for Bin Packing," *J. Computer and System Sciences,* vol. 8, pp. 272-314, 1974.


Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.

**Nikolaos D. Doulamis** (S'96-M'00) received the Diploma (with the highest honor) and the PhD degree in electrical and computer engineering from the National Technical University of Athens (NTUA) in 1995 and 2000, respectively. His PhD thesis was supported by the Bodosakis Foundation Scholarship. He joined the Image, Video, and Multimedia Laboratory of NTUA in 1996 as a research assistant. From 2001 to 2002, he served his mandatory duty in the Greek army at the Computer Center Department of the Hellenic Air Force. Since 2002, he has been a senior researcher at NTUA. His research interests include video transmission, content-based image retrieval, summarization of video sequences, and intelligent techniques for video processing. He was awarded as the Best Greek Student in the field of engineering in the national level by the Technical Chamber of Greece in 1995. In 1996, he received the Best Graduate Thesis Award in the area of electrical engineering with A.D. Doulamis. During his studies, he has also received several prizes and awards from the NTUA, the National Scholarship Foundation, and the Technical Chamber of Greece. In 1997, he was given the NTUA Medal as Best Young Engineer. In 2000, he served as the chairman of the technical program committee of the VLBV '01 workshop, and he has also served on the program committee of several international conferences and workshops. In 2000, he was given the Thomaidion Foundation Best Journal Paper Award in conjunction with A.D. Doulamis. He is an editor in the *Who is Who* bibliography. He is a reviewer of IEEE journals and conferences, as well as other leading international journals. He is a member of the IEEE.

**Anastasios D. Doulamis** (S'96-M'00) received the Diploma (with the highest honor) and the PhD degree in electrical and computer engineering from the National Technical University of Athens (NTUA) in 1995 and 2000, respectively. His PhD thesis was supported by the Bodosakis Foundation Scholarship. From 1996 to 2000, he was with the Image, Video, and Multimedia Laboratory of NTUA as a research assistant. From 2001 to 2002, he served his mandatory duty in the Greek army at the Computer Center Department of the Hellenic Air Force. In 2002, he joined the NTUA as senior researcher. Since September 2006, he has been an assistant professor at the Technical University of Chania. His research interests include nonlinear analysis, neural networks, multimedia content description, and intelligent techniques for video processing. He has received several awards and prizes during his studies, including the Best Greek Student in the field of engineering in the national level in 1995, the Best Graduate Thesis Award in the area of electrical engineering with N.D. Doulamis in 1996, and several prizes from the NTUA, the National Scholarship Foundation, and the Technical Chamber of Greece. In 1997, he was given the NTUA Medal as Best Young Engineer. In 2000, he received the Best PhD Thesis Award, in conjunction with N.D. Doulamis, from Thomaidion Foundation. In 2001, he served as technical program chairman of the VLBV '01. He has also served on the program committee of several international conferences and workshops. He is a reviewer of IEEE journals and conferences, as well as other leading international journals. He is the author of more than 100 papers in the above areas in leading international journals and conferences. He is a member of the IEEE.



**Emmanouel A. Varvarigos** received the Diploma in electrical and computer engineering from the National Technical University of Athens in 1988 and the MS and PhD degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, Massachusetts, in 1990 and 1992, respectively. His research interests include protocols and algorithms for high-speed networks, all-optical networks, high-performance switch architectures, grid computing, parallel architectures, performance evaluation, and ad hoc networks. In 1990, he worked as a researcher at Bell Communications Research, Morristown, New Jersey. From 1992 to 1998, he was an assistant and later an associate professor in the Department of Electrical and Computer Engineering, University of California, Santa Barbara. From 1998-1999, he was an associate professor in the Department of Electrical Engineering, Delft University of Technology, the Netherlands. In 1999, he became a professor in the Department of Computer Engineering and Informatics, University of Patras, where he is currently the director of the Communication Networks Laboratory. He is also the director of network technologies sector of the Research Academic Computer Technology Institute (RA-CTI). He was the organizer of the 1998 Workshop on Communication Networks and was on the program committee of several international conferences.

**Theodora A. Varvarigou** (S'88-M'92) received the BTech degree from the National Technical University of Athens, Athens, in 1988, the MS degrees in electrical engineering and computer science from Stanford University, Stanford, California, in 1989 and 1991, respectively, and the PhD degree from Stanford University also in 1991. She worked at the AT&T Bell Laboratories, Holmdel, New Jersey, between 1991 and 1995. Between 1995 and 1997, she worked as an assistant professor at the Technical University of Crete, Chania, Greece. She is an associate professor at the National Technical University of Athens. Her research interests include parallel algorithms and architectures, fault-tolerant computation, optimization algorithms, and content management. She is a member of the IEEE.

▹ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: University of Patras. Downloaded on January 29, 2009 at 05:21 from IEEE Xplore. Restrictions apply.