CHAPTER 16

Timed/Advance Reservation Schemes and Scheduling Algorithms for QoS Resource Management in Grids

Emmanuel Varvarigos
University of Patras, Department of Computer Engineering and Informatics, Patras, Greece

Nikolaos Doulamis, Anastasios Doulamis, Theodora Varvarigou
National Technical University of Athens, Department of Electrical and Computer Engineering, Athens, Greece

CONTENTS

ISBN: 1-58883-038-1

Engineering the GRID
Edited by B. Di Martino, J. Dongarra, A. Hoisie, L. T. Yang, and H. Zima
Pages (1–23)

1. INTRODUCTION

The Computational Grid is a fundamentally new computational paradigm, which may result, if successful, in a revolutionary explosion in the level of computational power that can be applied to a given parallelizable set of tasks [1]. This level of computational power is well beyond the reach of any single computer system, whether a common PC or a massively parallel supercomputer. Various *ad hoc* implementations have already successfully illustrated the Grid's potential. However, its true global success requires the development of thoroughly specialized supporting technologies, especially for the management of the grid resources. Unfortunately, the Grid's apparent constitution as a combination of already well-developed and mature technologies is, in fact, quite misleading [2].

The Computational Grid is both (1) a way to share resources and (2) a way to do things faster than a single computer can do. The critical resources in the Grid are the computational power, memory, communication bandwidth, and devices of all kinds that have to be shared [1]. Clearly, it is these resources that will account for the largest fraction of its cost. The remaining cost, which is mainly the cost of the application software to be used, is expected to decrease (to be amortized) as the number of users increase—and there is no great penalty in distributing this software everywhere in the grid. The true waste occurs when the computational and the communication resources that are shared are not efficiently used. Therefore the development of protocols and algorithms for the efficient management of resources is a key to the success of grid computing [3].

Computers running a task on a Grid have to get the required code and data from a remote machine; also, when doing computations, machines often have to exchange intermediate results. In many cases, the time required for communication is more than the time spent by machines on computations. The success stories of the Grid are currently limited mainly to coarse grain computation. If Grids are eventually to be used for fine grain computations, communication delays will become a performance bottleneck. It is therefore important that the Grid resource manager takes communication delays into account when reserving resources and scheduling tasks. It also has to consider the interdependence between communication and computation tasks in order to optimise the performance through the use of pipelining techniques and clever scheduling.

The Grid has to be able to provide an agreed upon *Quality of Service* (QoS) to its users. Without it, the *users* may be reluctant to pay for Grid services or contribute resources to Grids, which would hinder its development and limit its economic significance. The need for QoS has been confirmed by the Global Grid Forum (GGF) in the special working group dealing with the area of scheduling and resource management [4]. New abstractions and concepts should be introduced at the architecture level to allow users to make service level agreements (SLAs) with the Grid, and new capabilities are required to enable the enforcement of such SLAs.

The resource manager of a Grid uses information about the job characteristics and requirements to determine when and on which processor each job will execute. The objective of the resource manager is to assign resources to tasks in an efficient and fair way, while meeting to the degree possible the QoS requirements of the tasks. Efficiency in the use of resources is clearly important because this is what motivated the Grid in the first place. Fairness is important because it is inherent in the notion of sharing, which is the raison d'etre of the Grid. Meeting the requirements of the users is important because otherwise the users will not want to use, pay, or contribute resources to the Grid [4].

In order to provide the agreed QoS to the users, the resource manager has to be able to reserve (parts of) resources for the execution of specific tasks. The requirement of on demand and efficient use of resources implies that resources should be allocated to a task only for the time interval during which they are actually used, and should be available to other tasks for the remaining time. This is not accomplished by existing resource reservation protocols, which usually reserve resources for considerably more time than the minimum required. To alleviate this problem, we introduce in Section 3 a new resource reservation protocol, called the TARR protocol, which uses the concepts of *timed and advance reservation* of resources. The protocol uses estimates of the communication delays and the task execution times in order to reserve (computational and communication) resources only for the exact time periods during which they will actually be used by a task, and leave these resources free to be used by other tasks for the remaining of the time.

We also consider the QoS scheduling problem, whose goal is to assign the tasks requesting service to the available processors so as to satisfy their time constraints. The time constraints of a task include the task's deadline (that is, the time by which it is desirable for it to complete execution) and the task's earliest starting time on each processor (that 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.

Meeting the QoS requirements of one user should not be achieved by sacrificing the QoS of another user. When the desired QoS cannot be provided, the degradation in the QoS provided should be graceful to all users. This naturally leads to the notion of *fairness*, which is an important issue we address in the second part of this paper. Fairness does not mean that all users have equal access to Grid resources. Instead, our notion of fairness is based on the max-min fair share concept, and it takes into account the QoS requirements of the users, and also a weighting parameter that is determined by economic considerations. The weighting parameter permits us, in case of congestion, to treat more favourably (in a quantifiable way) users who are willing to pay more for the service they get or who have contributed more resources to the common Grid infrastructure, or who have to be treated preferentially for some other reason. We propose two new scheduling algorithms, which we call the Simple Fair Task Ordering (SFTO) scheme, and the Adjusted Fair Task Ordering (AFTO) scheme, which differ in the degree of fairness they provide but also in their implementation complexity. When there is no congestion, each user gets the requested QoS, but when congestion arises (meaning that the desired QoS cannot be delivered to all users), the QoS that each user receives (as measured by the computational power he is given, or his completion time, or the amount of time by which he misses his deadline) is degraded in a way determined by his requested QoS and the weighting factor.

The remainder of the paper is organized as follows. In Section 2, we describe the task characteristics that are important for resource management. In Section 3 we present the TARR protocol for resource reservation, designed to make efficient use of the available resources. Starting with Section 4 we turn our attention to the scheduling problem, whose purpose is to decide the processor and the time interval on which each task should be executed. We first introduce some basic notation and then give the formulation on which our algorithms will be based. In Section 5, we introduce the notion of fair completion time and propose two Grid scheduling algorithms that take fairness into account. Experimental results and comparisons with traditional scheduling schemes, such as the Earliest Deadline First (EDF) and the First Come First Served (FCFS) schemes are presented in Section 6. Finally, Section 7 concludes the paper.

2. TASK CHARACTERISTICS IMPORTANT FOR RESOURCE MANAGEMENT

In order to perform its function the resource manager needs to consider the following task characteristics:

- *The estimated workload of the task.* The workload can be categorized based on the type of system resource we are referring to: For computer resources, the workload can be defined, as the number of run-time instructions of the task. For network resources, the workload can be defined as the number of bits that have to be transferred. For storage resources, the workload can be measured by the number of bits stored and the duration of time for which they have to be stored.

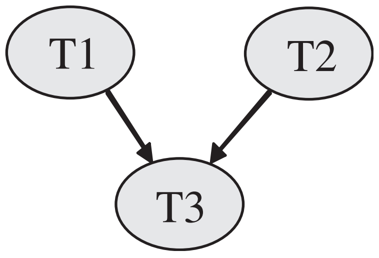


Figure 1. The interdependence between the tasks can be given in terms of a DAG.

- *The variance of workload.* Since the workload is not generally known a priori and is better modeled as a random variable, it is useful for the resource manager to have a measure of the variability of the workload around its mean.
- *The required Quality of Service.* The definition of QoS might include requirements realted to the task deadline (i.e., the required completion time of the task), the probability to miss the deadline and the reliability. For example, fault tolerance is important, the task may have to be scheduled on more reliable resources or on more than one processor for execution.
- *The interdependence between the tasks.* Any temporal relations between the tasks could be given in the form of a directed acyclic graph (DAG), describing precedence constraints for task execution. For example, the DAG shown in Fig. 1, states that Tasks T1 and T2 must be executed before Task T3. The resource manager must be able to schedule the tasks in the proper order.
- *The price that the user is willing to pay.* Depending on the cost that the user is willing to incur, the scheduler may send the tasks to more or less expensive resources [3]. Also, in case some tasks have to be rejected, the price that a user pays will influence the choice of tasks that are rejected. The cost of a user is not necessarily an explicit amount that is charged. Instead it may be implicitly found from the resources the user is contributing to the Grid infrastructure. In Section 5 we propose scheduling algorithms that use the max-min fair share concept to take into account (through the introduction of a weighting parameter) such economic considerations.

It is important to point out that some or all of the task characteristics mentioned above may not be known to the resource manager in advance. In that case the resource manager must still be able to work properly and make the best decision with the information that is available at the time the decision is made.

3. TIMED AND ADVANCE RESERVATIONS

In this section we propose a resource reservation scheme for Grids that uses the notions of advance and timed reservations. To better appreciate the advantages these two features offer, we first describe the usual approach where advance/timed reservations are not used. We refer to such a scheme as a *standard resource reservation scheme*, and we explain the reasons such schemes are inefficient.

3.1. Standard Resource Reservation Schemes

In standard resource reservation schemes, when the scheduler assigns a task to a resource, it records a reservation for that resource in its database. The resource is considered allocated to the task for an unspecified amount of time, starting at the time the the task-to-resource assignment is made. As far as the scheduler is concerned, the resource remains booked for the task until the scheduler is informed about the task completion. This situation is shown in Fig. 2, where the scheduler waits for a *release message* to arrive from the resource before it can allocate the resource to a new task. To allocate the resource to a new task, the scheduler must send an *allocation message* to the application telling the user-side where

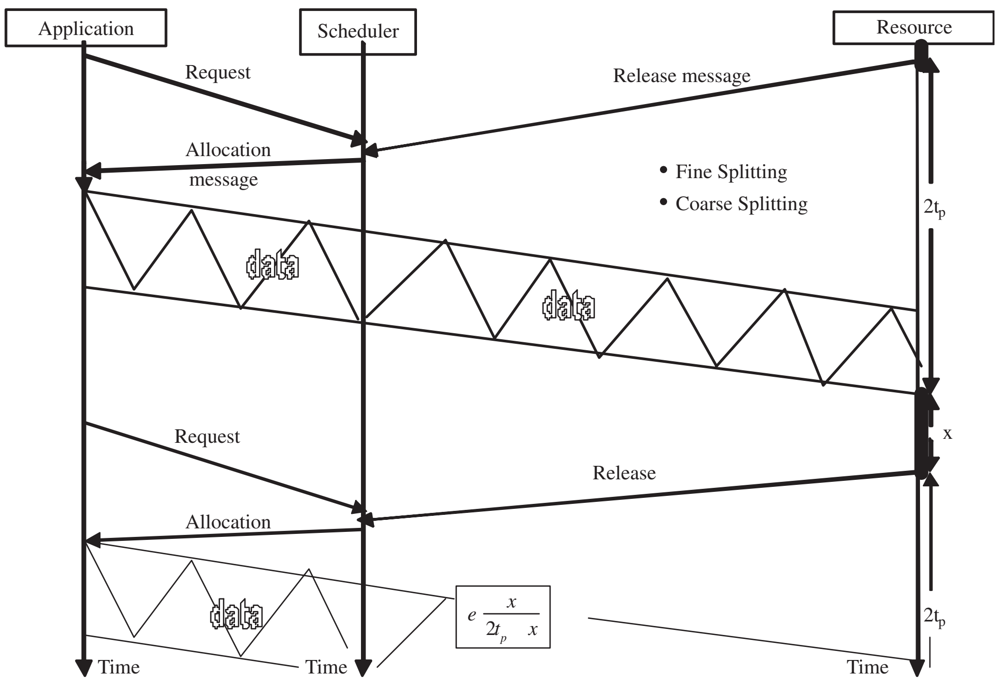


Figure 2. Possible scenario when a standard resource reservation scheme is used.

the data and code required for the execution of the task should be transferred. The data and code are then transferred to the resource, which takes some additional communication time. During the transmission of the allocation message and during the transmission of the data, the resource remains (unnecessarily) inactive. When all the data have arrived at the resource, the task begins execution. When the task is completed, the resource remains again (unnecessarily) inactive until the release message arrives at the scheduler, which can then allocate it to another task.

We denote by $2t_p$ the average time that elapses between the time a resource sends the release message to the scheduler to inform it about the completion of a task and the time all the data required to execute the next task arrive at that resource. We also denote by x the mean execution time of a task on that resource. It is then clear that when a standard resource reservation scheme is used, the efficiency with which a resource is used is at most

$$e = \frac{x}{2t_p + x} \tag{1}$$

Note that the efficiency factor e may be considerably smaller than 1, and it decreases as x decreases or as $2t_p$ increases ($2t_p$ is at least as large as the roundtrip propagation delay). In order for the Grid to be useful for a number of different applications, we would like to be able to use fine grain computation (where x is small) and also be able to use remote resources (where $2t_p$ is large), both of which correspond to small values for the efficiency factor e. Thus, standard resource reservation algorithms fundamentally restrict the efficiency with which Grid resources are used. In the following subsection we show how we can use timed/advance reservations to make the efficiency factor e close to 1, independently of the communication delays in the network and the granularity of the tasks.

3.2. The Timed/Advance Resource Reservation (TARR) Scheme

In the Timed/Advance Resource Reservation scheme (abbreviated, TARR) that we propose, the scheduler maintains an *utilization profile* for each resource, which keeps track, as a function of time, of the future utilization of that resource (see Fig. 3). More specifically,

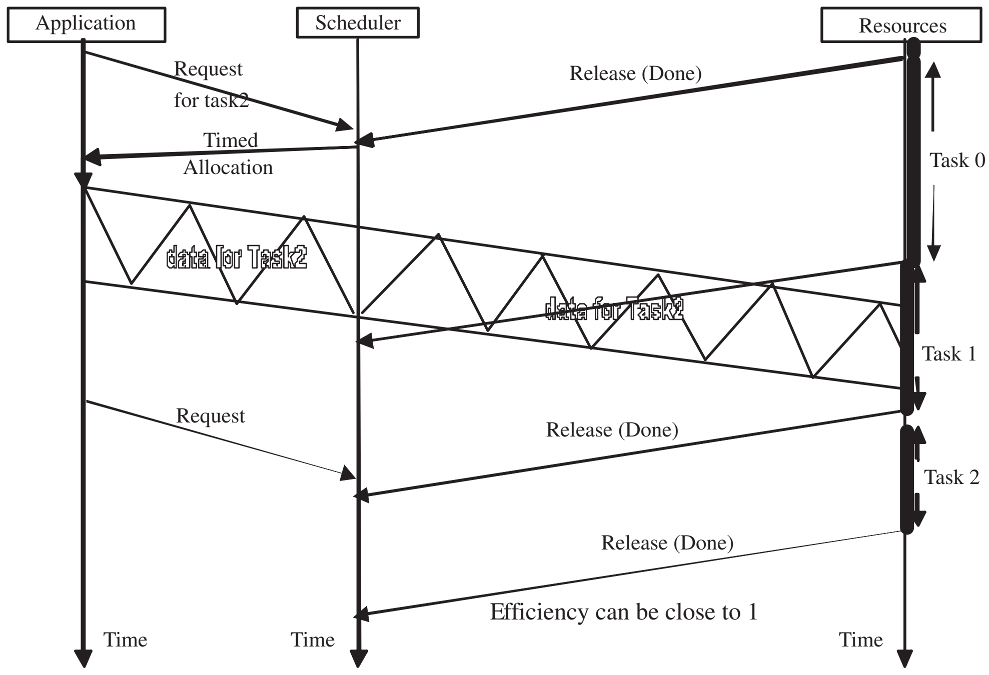


Figure 3. Resource reservation when advance/reservations are used. The resources are reserved only for the time during which they are actually used by a task.

the utilization profile of a computational resource records the (future) time intervals for which the resource has already been reserved for executing tasks. (Note that the resource may also be the communication bandwidth on a link, in which case the utilization profile keeps track of the bandwidth that has been reserved on that link as a function of time, or it may be a memory resource or an instrument that is shared by users). When a resource is released, the scheduler is informed in order to keep this information up to date. When a request for a task is sent to the scheduler, the scheduler allocates a resource to the task for a specific future time interval. In calculating the start time (ST) of that time interval the scheduler takes into account the estimated communication delays, so as to make sure that all the data necessary to run the task will have arrived at the resource before time ST. In calculating the end time (ET) of the interval the scheduler takes into account the ST and the estimated workload of the task. Therefore, the scheduler uses the resource utilization profile to allocate a future interval for each task scheduled on a given resource, so that when one task is completed, the data for the next task, are already available at the resource for the new task to start execution immediately. This maximizes the efficiency of the resources and the efficiency factor e can get very close to 1.

An example of a timed/advance reservation made using the resource utilization profile is shown in Fig. 4. The scheduler is responsible for assigning the tasks, each of which has a known (estimated) workload, to the appropriate resource using the utilization profiles of the resources. The information that has to be provided to the scheduler to make a decision is:

- the required CPU capacity (CPU)
- the required execution time (x) assuming capacity equal to CPU is allocated
- the job completion deadline (D)
- the communication delays δ_{ij} for the transfer of all the data involved in the execution of task T_i from the user to resource j.

These parameters are supplied to the scheduler by the user and possibly by a forecasting tool. The total estimated workload can be calculated as $WL = CPU * x$. The scheduler takes into account the communication delay for transferring data from the user application to a resource, in order to find the earliest possible starting time of a task at that resource.

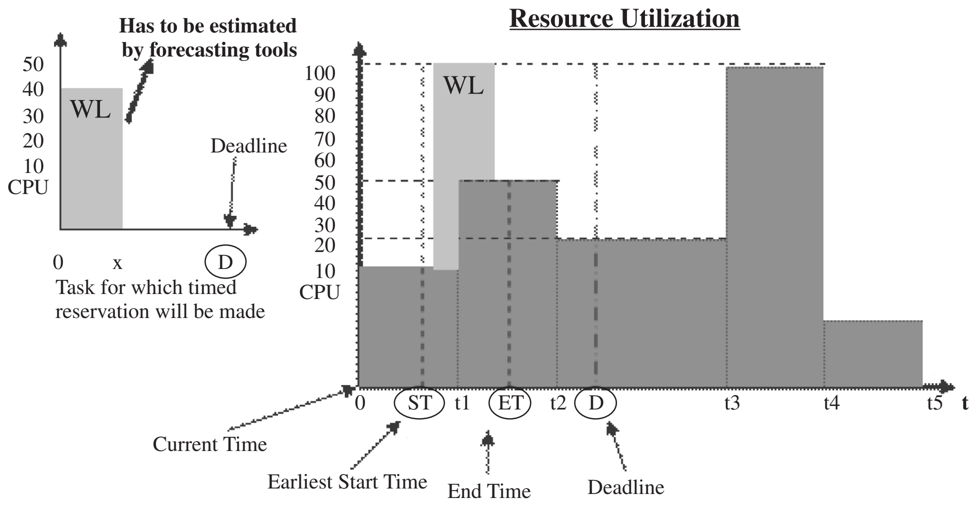


Figure 4. Illustrates an example of a timed/advance reservation using the resource utilization profile. The start time ST of a reservation interval is chosen so as to ensure that all data necessary for the execution of the task have arrived to that resource by time ST. The end time ET of a reservation interval is chosen so as to ensure that the task is completed by time ET. If we cannot allocate resources so that $ET < D$, where D is the deadline the job is rejected.

The next scheduling step is to reserve (a portion of) the resource for a specific duration for the execution of task. The reservation is performed by fitting the total workload in the resource utilization profile. This step may also involve a negotiation process, as shown in Fig. 5. It is possible that the CPU capacity required for a task may not be available at the time it is requested, in which case a reservation for a time interval later than the time requested may be chosen. Also, the scheduler may negotiate/specify new parameters for the CPU capacity and a corresponding time duration that will result in the same total workload. When the reservation is completed, the estimated end time of the task is calculated and the resource utilization profile is updated.

In the resource utilization profiles shown in Figs. 4 and 5 the local resource manager is assumed to support time sharing, so that it can allocate fractions of the CPU capacity to the

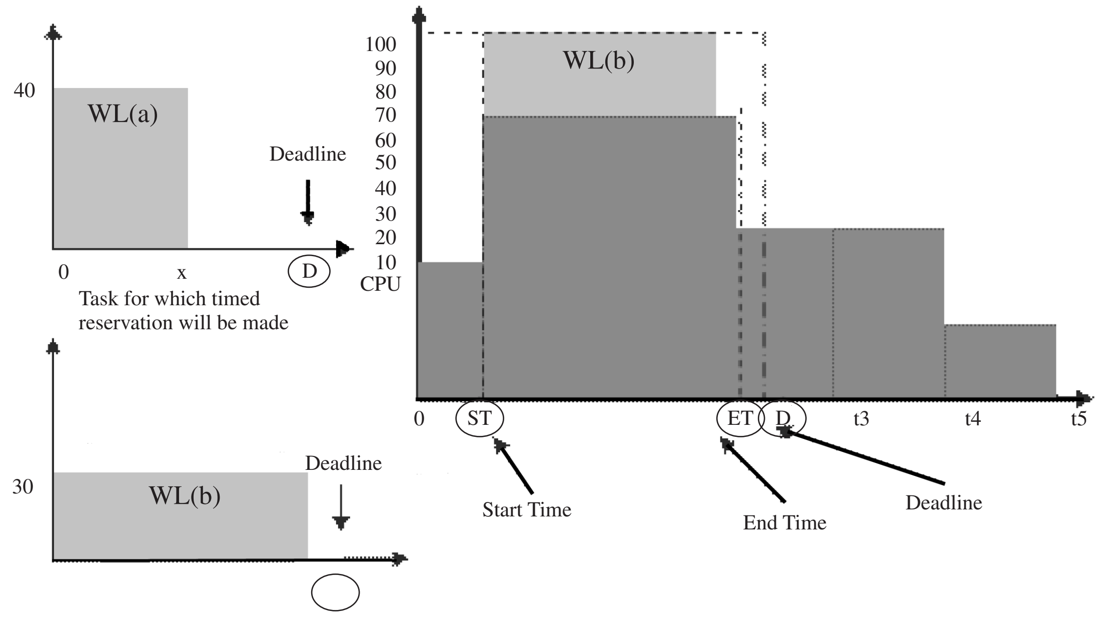


Figure 5. Illustrates a negotiation example. The applications requests 45 units of CPU capacity for duration equal to t, but gets 30 units of CPU capacity for duration equal to $3t/2$ after the negotiation. (a) The workload as it was supplied by the application; (b) The workload as it was specified after the negotiation process.

tasks. If this is not the case, and the resource has to be allocated a 100% of the time time at a task, the utilization profiles and the start and end times for the reservation intervals can be easily modified.

3.3. Timed/Advance Reservations Under Uncertainty

In the examples considered so far it was assumed that the task workload and the communication delays can be estimated with good accuracy by the forecasting tools and/or by the user. In practice, however, these estimates may not be accurate; the scheduler should be able to function correctly in the presence of such uncertainty.

Figure 6 shows the utilization profile at a resource where a reservation for a given task has to be made. We assume that some rough estimates for the task workload and the communication delays involved are available, but there is a certain degree of uncertainty in these estimates. The degree of uncertainty can be expressed in terms of variance. Since the communication delays are not accurately known, it is not known when exactly the data required for the execution of the task will arrive at the resource. Also, since the workload is not accurately known, it is not known how much time it will take for the task to complete after it starts execution.

The rule to be followed by the scheduler in order to work correctly in the presence of such uncertainties is that the scheduler must underestimate the start time of the reservation (i.e., start the reservation earlier than its estimate) and overestimate the end time of the reservation (i.e., record the end time later than its estimate) by some small multiple of the standard deviation of these estimates. For example, if the mean transfer time for data is estimated to be δ seconds, the actual reservation on the resource can be made starting at time $\delta - 3\sigma_\delta$ after the present time. In Fig. 6 the estimated start time is shown as ST, while the time the reservation actually starts is shown as UST (Underestimated Start Time). A similar rule is used when the workload is not known with accuracy. In that case the end time of the task has to be overestimated, in order to make sure that the task will have all the required resources for it to complete execution. The end time OET (Overestimated End Time) used for the actual reservation is calculated by overestimating the workload ($OWL = WL + 3\sigma_{wl}$).

The rules mentioned above can be used separately or together depending on the accuracy the forecasting tools can provide. For example, when the the workload estimate is precise but the communication delays cannot be accurately estimated, we need only underestimate the start time of the reservation.

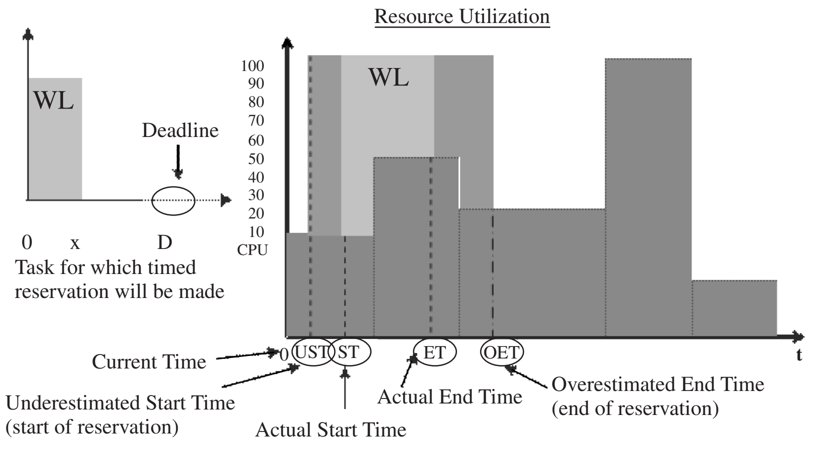

Resource Utilization
WL
Deadline
0
x
D
Task for which timed reservation will be made
100
90
80
70
60
50
40
30
20
10
CPU
WL
0
UST
ST
ET
OET
t
Current Time
Underestimated Start Time (start of reservation)
Actual Start Time
Actual End Time
Overestimated End Time (end of reservation)

Figure 6. Illustrates the way timed/advance reservations can be made in the presence of uncertainty. The rule that should be followed is that, when in doubt, we should overestimate the workload and/or underestimate the start time in order to be safe. For example, when the variance σ_{wl}^2 of the workload is known, we can assume that the workload is less than $WL + 3\sigma_{wl}$ with high probability. Similarly, a lower bound that could be used for the start time is $ST - 3\sigma_\delta$, with high probability.

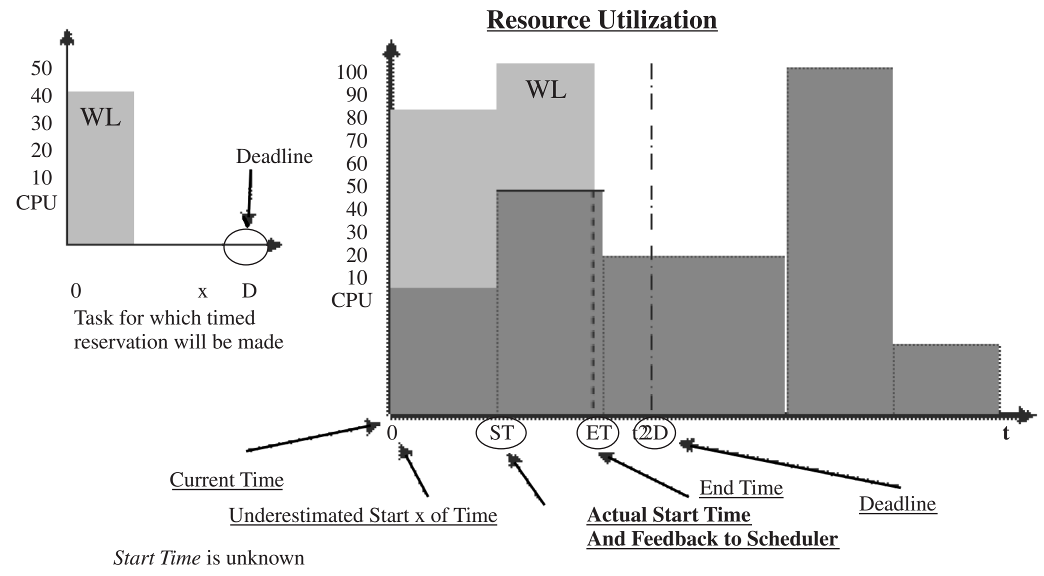


Figure 7. Advance reservations when the start time is unknown.

When the forecasting tools are unable to provide any estimates of the communication delays involved (or they provide no measure of how accurate the estimates are, that is, they provide no σ_δ), then the reservation should start immediately, as shown in Fig. 7, as if the delay for the data to be transferred from the application to the resource were zero. In that case, the resource is reserved much earlier than required, and efficiency decreases. In Fig. 7 the actual start time is shown with ST and the task deadline is shown with D. If the workload is known, then when the task actually starts execution, a message is sent to the scheduler to allow the calculation of the end time, which is then recorded in the utilization profile.

When no estimate for the total workload is available, the end time should be considered infinite. The resource is then released when the task has completed execution and a release message has arrived to the scheduler (Fig. 8). The efficiency factor e is then the same with that of standard resource reservation protocols. Clearly, when the workload is not known, the task deadline cannot be guaranteed, and the user must be informed in advance that such no guarantee og the QoS can be provided.

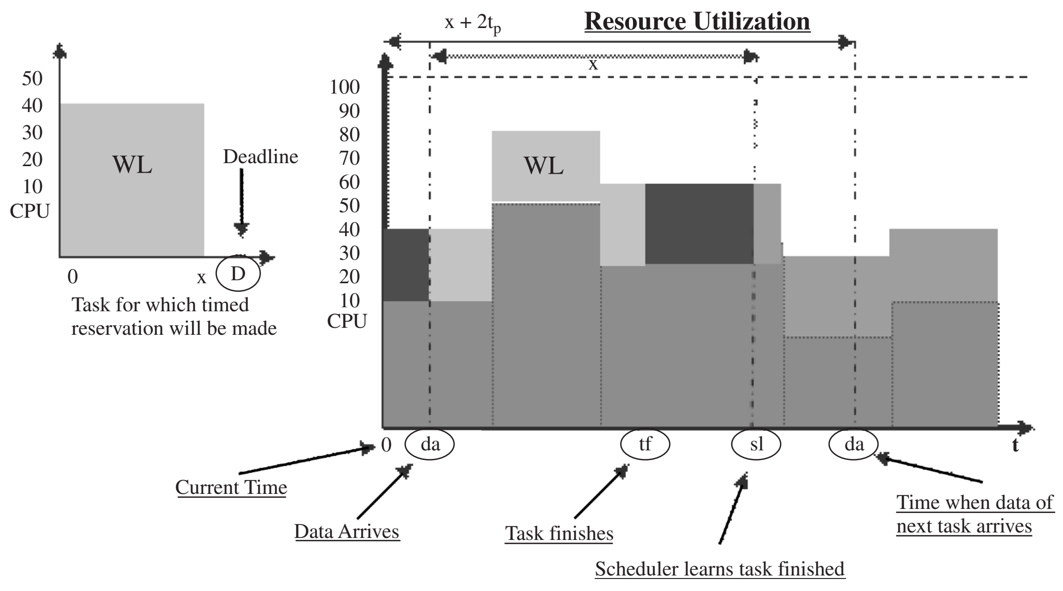


Figure 8. Advance reservation when no estimates for the workload and the communication delays are available.

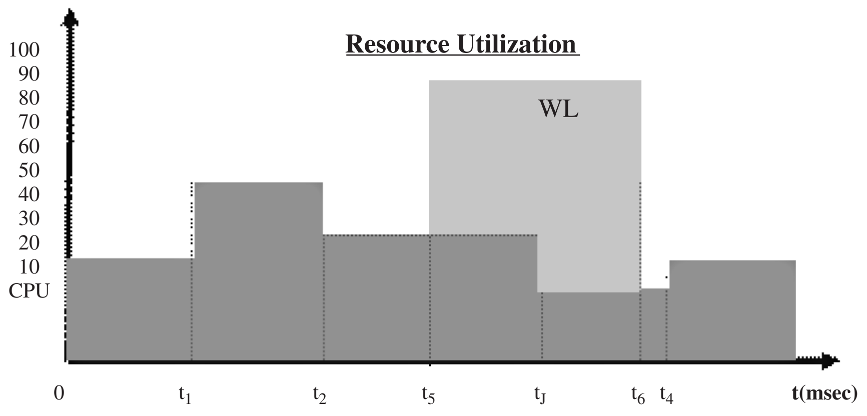


Figure 9. Illustrates the data structure used for recording the utilization profile of a resource. The data structure is essentially a linked list of pairs $(r_0, 0), (r_1, t_1), (r_2, t_2 - t_1), \ldots, (r_n, t_n - t_{n-1})$ used to record the discontinuities in the utilization profile. The horizontal axis records time relative to the present time: every msec the vertical axis moves to the right by 1 msec. The list update algorithm is simple and requires a small number of operations. The introduction for a new task between times t_5 and t_6, as is shown in the figure, can be easily done by inserting two new elements in the list.

3.4. Data Structures Required by the TARR Protocol

The TARR scheme requires the scheduler to keep track of the utilization profile of each resource. This can be easily done using a linked list, each element of which is a pair of numbers, as described in the caption of Fig. 9. When a reservation for a new task is made, two new elements (pairs) are inserted in the list, corresponding to the times the reservation for the new task starts and ends, together with the CPU capacity reserved for that task. The first element $(r_0, t_0 = 0)$ of the list always refers to present time. As the time passes, the second pair (r_1, t_1) is changed by periodically updating the time field t_1. For example, if the periodicity (time granularity) of the updates is 1 msec and $t_1 = 5$ msec, then after 1msec t_1 will be changed to 4 msec since t_i refers to times relative to t_{i-1}, the time fields t_i, for $i > 1$ need not be updated. The period of updates of the utilization profile should be decided based on the expected granularity of the tasks. For example, a msec time scale should be used when the typical execution time for a task is of the order of few 10s of msec.

4. SCHEDULING MULTIPLE TASKS ON RESOURCES

The TARR protocol described in the preceding section gives an efficient way for reserving resources in a grid, using the concept of timed/advance reservations. However, we have not yet discussed the way the resources where the tasks are assigned are chosen. Starting with this section, we focus on this problem, called the *scheduling problem*, and propose algorithms to solve it in a way that is both efficient and fair.

Figure 10 illustrates an example of the scheduling problem where we assume that the TARR protocol is used for making reservations. In this example, there are four tasks that have been submitted to the scheduler, and the grid consists of only two resources (CPUs) with utilization profiles as shown in this figure. The scheduler has to decide the resource that should be allocated to each task and the corresponding time interval that should be reserved. Figure 10 assumes, for the sake of an example, that an Earliest Deadline First rule, discussed in Section 4.3, is used; more intelligent scheduling algorithms will be considered in the following sections.

We will say that a task is *non-preemptable* if once it starts execution on a processor, it has to be completed on that processor without interruption. In the scheduling algorithms we will propose we assume that tasks are non-preemptable [5]. This is the most limiting case, since any scheduling algorithm that works with non-preemptable tasks also works when this constraint is removed giving the scheduler more freedom. Task preemptions and interruptions are in any case undesirable, since they imply considerable overhead for transferring data to other processors, saving process state information, and so on. We

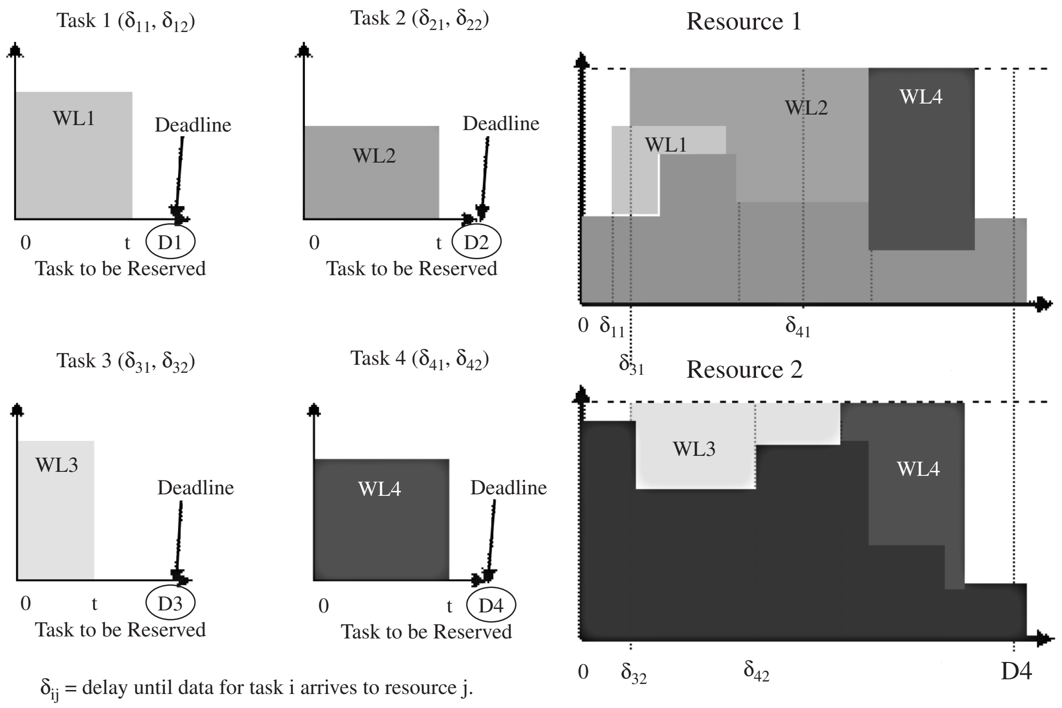


Figure 10. Example of scheduling tasks on resources when the TARR protocol is used for making the reservations. In this example, the scheduler considers the tasks by first sorting them according to the EDF rule described in Section 4.2. The algorithm first tries to fit Task 1, which is the the task with the earliest deadline to the Resource 1. In this example this is feasible, so the reservation is made using the TARR protocol, and the scheduler updates the utilization profile of the Resource 1. The scheduler then considers Task 2, which it finds can also be assigned to Resource 1. Task 3, which is considered next, cannot be allocated to Resource 1, so the scheduler attempts to assign it to Resource 2. Task 3 fits in the utilization profile of Resource 2, even though it will take longer for the data required to execute the task to reach that resource (the delay δ_{31} is smaller than δ_{32}). In this example, we assume that the Task 4 is sent to both resources, because, for example, it has higher fault tolerance requirements than the other tasks. In all cases the scheduler makes sure that all tasks are scheduled for completion before their deadline. If this were not possible, the scheduler would inform the user of the possible violation of the deadline, and let the user the option of either withdrawing the task, or accepting a new deadline, or resubmitting the task with new QoS requirements.

also assume that when a task is executed on a machine it occupies 100% of its computational power, that is, time sharing is not supported. We consider this assumption reasonable since (a) local, non-grid jobs can be accounted for by excluding from the machine's computational capacity the fraction used by these jobs, and (b) the ability to execute multiple grid jobs simultaneously on a machine cannot improve performance (the processor sharing discipline is the worse performing among work-conserving disciplines), and (c) any schedule obtained under this assumption is also feasible under the assumption that time-sharing is available.

4.1. Previous Work on Scheduling Algorithms

Several scheduling algorithms have been reported in the literature so far. The most well-known scheduling algorithm is the Earliest Deadline First (EDF) [6], which assigns the highest priority to the task with the most imminent deadline. Another scheduling approach is the Least Laxity First (LLF) algorithm [7, 8], where the tasks are selected for execution in order of non-decreasing slack time, defined as the difference between the task's relative deadline and its remaining computational time. A simple rule to determine the processor on which a task is executed is the Earliest Start Time (EST) rule. The earliest start time 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. New algorithms for resource reclaiming from precedence constrained tasks in multiprocessor real-time systems have been proposed in [9]. In case of multiprocessor systems the authors in [10] have proposed an integrated heuristic that takes into account both the task deadline and earliest start time and performs better than the EDF, LLF and MPTF algorithms. Several heuristic scheduling algorithms for the multiprocessor case were also proposed in [11]. The scheduling algorithms mentioned above assume that the tasks are non-preemptable. Scheduling algorithms dealing with preemptable tasks have also been reported in the literature [12–14], where 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 non-preemptable tasks, but this increase in schedulability is obtained at the expense of a higher implementation overhead.

Scheduling algorithms for grid computing systems have also been recently proposed. An economic modeling of the scheduling problem was given in [3] using the concepts of commodities and auctions. In this work, a task whose owner is willing to pay more for its execution has a higher probability of being scheduled compared to the remaining tasks. A similar micro-economic approach has aslo been proposed in [15]. Three approaches for grid scheduling that minimize the task earliest completion time are presented in [16]. A new multisite scheduling algorithm for the GrADS grid infrastructure is giving in [17]. Finally, performance evaluation of grid systems has been reported in the work of [18].

The previously mentioned scheduling algorithms do not deal with congestion, and do not say what happens when the tasks cannot all be feasibly scheduled. As a result, fairness considerations also play no role in these algorithms. Fair scheduling schemes have been extensively studied in the networking literature, especially for scheduling packet flows over Internet routers [19, 20]. Some of the ideas we will use for fair QoS scheduling in Section 5 are drawn from concepts introduced in the related networking literature.

4.2. 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, $i = 1, 2, \dots, N$, 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, etc., [21]. An algorithm for workload prediction of 3D rendering in a Grid architecture is presented in one of our earlier works in [22]. We assume the tasks are non-preemptable, 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% of the processor capacity.

We assume that there are M processors in the system, and the computation capacity of processor j is equal to c_j units of capacity. (The computation capacity of a processor is taken to be 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

$$C = \sum_{j=1}^{M} c_j \tag{2}$$

That is, if the Grid could be viewed as a single computer it would have computation capacity equal to C.

Let d_{ij} be the communication delay between user i and processor j. More precisely, d_{ij} is (an estimate of) the time between the time a decision is made by the scheduler to assign task T_i to processor j, and the arrival of all files necessary to run task T_i to processor j. We assume that the communication delays d_{ij} are known to the scheduler, and they are provided by a prediction mechanism that takes into account the size of the files that have to be transferred, the available bandwidth and propagation delays in the network, and statistical data.

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 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 γ_j be the estimated completion time of the tasks already running on or already scheduled on processor j. γ_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, γ_j corresponds to the remaining time until completion of the tasks already allocated to processor j. We define the *earliest starting time* of task T_i on processor j as

$$\delta_{ij} = \max(d_{ij}, \gamma_j) \tag{3}$$

δ_{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

$$\delta_i = \frac{\sum_{j=1}^{M} \delta_{ij} c_j}{\sum_{j=1}^{M} c_j} \tag{4}$$

We will refer to δ_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 $\sum_{j=1}^{M} c_j$. Since in a Grid computation power is distributed, δ_i plays a role reminiscent of that of the (mean) memory access time in uni-processor computers.

In the fair scheduling algorithms 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

$$X_i = \frac{w_i}{D_i - \delta_i} \tag{5}$$

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 δ_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 Section 5 attempt to degrade the QoS experienced by the tasks (as measured by the computation rate allocated to the task, or the amount of time by which the deadline is missed as a percentage of $D_i - \delta_i$) in a (weighted) fair way.

The scheduling algorithms that we will propose 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*).

4.3. Earliest Deadline First and Earliest Completion Time Rules

The most widely used urgency-based scheduling scheme is the Earliest Deadline First (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 (i.e., 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 Earliest Completion Time (ECT) technique presented next can be used.

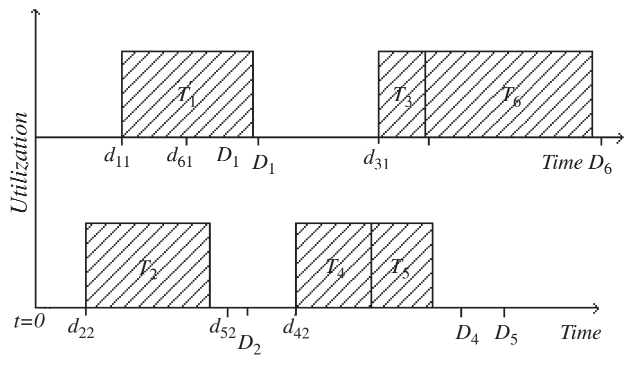


Figure 11. An example of the Earliest Deadline First/Earliest Completion Time (EDF/ECT) algorithm for the case γ_j is defined as the processor release time. In this figure, we assume that all tasks T_i, $i = 1, 2, \ldots, 6$, request service at time $t = 0$, the processors have equal computational capacity ($c_1 = c_2$), 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 tie). Task T_2 is then assigned for execution on processor 2 (since it is the processor that yields the earliest completion time). In a similar way, we assign the remaining tasks.

If task T_i starts execution on processor j at the earliest starting time δ_{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% of a processors 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

$$\hat{j} = \arg \min_{j \in \{1 \cdots M\}} (s_{ij} + w_{ij}) \tag{6}$$

The earliest starting time δ_{ij} depends through Eq. 3 on the time γ_j at which the last task already allocated to processor j is expected to complete service.

A note regarding the way γ_j is defined is necessary here. One way to define γ_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. Figure 11 illustrates a scheduling scenario in which (a) the *task queuing order* is selected using the EDF algorithm (b) the *processor assignment* is selected using the ECT approach and (c) γ_j is defined as the processor release time.

Defining γ_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. 11), 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, to assign the task to the corresponding time interval. Among all candidates time intervals the one that provides the earliest completion time is selected. Figure 12 shows how the schedule for the example given in Fig. 11 is improved by exploiting capacity gaps. The completion times of tasks T_5 and T_6 are shorter than those obtained in Fig. 11.

5. FAIR SCHEDULING

The scheduling algorithms described in the preceding section 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 times 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

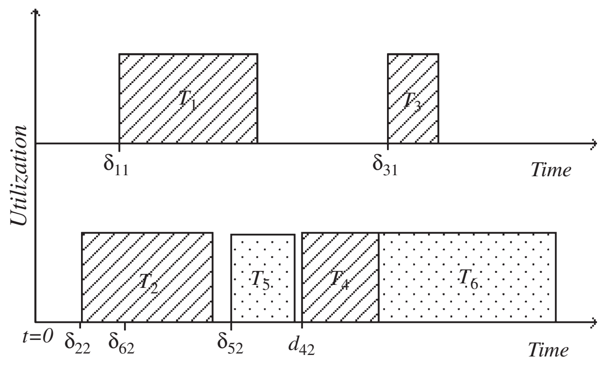


Figure 12. An example of the EDF/ECT algorithm that exploits processor utilization gaps.

according to 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 subsection.

5.1. Estimation of the Task Fair Rates

In our initial presentation, we assume that all tasks have equal weights, and all users are equivalent.

5.1.1. Ideal Non-Weighted 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.

The max-min fair sharing algorithm is described more formally as follows. The demanded computation rates X_i, $i = 1, 2, \dots, N$, 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 [Eq. 2]. 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 additional capacity $(C/N + (C/N - X_1)/(N - 1))$. This may be larger than what task needs T_2, 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 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 the *non adjusted fair computation rate* of the task T_i at the nth iteration of the algorithm. Then $r_i(n)$ is given by

$$r_i(n) = \begin{cases} X_i & \text{if } X_i < \sum_{k=0}^{n} O(k) \\ \sum_{k=0}^{n} O(k) & \text{if } X_i \geq \sum_{k=0}^{n} O(k) \end{cases}, \quad n \geq 0 \tag{7}$$

where

$$O(n) = \frac{C - \sum_{i=1}^{N} r_i(n-1)}{card(N(n))}, \quad n \geq 1 \tag{8}$$

with

$$O(0) = \frac{C}{N} \tag{9}$$

In Eq. 8, $N(n)$ is the set of tasks whose assigned fair rates are smaller than their demanded computation rates at the beginning of the nth iteration, that is,

$$N(n) = \{T_i : X_i > r_i(n-1)\} \quad \text{and} \quad N(0) = N \tag{10}$$

while the function $card(\cdot)$ returns the cardinality of a set. The process is terminated at the first iteration n_0 at which either $O(n_0) = 0$ or the number $card(N(n_0)) = 0$. The former case indicates congestion, while the latter indicates that the total grid computation capacity can satisfy all the demanded task rates, that is,

$$\sum_{i=1}^{N} X_i < C \tag{11}$$

The non-adjusted fair computation rate r_i of task T_i are obtained at the end of the process as

$$r_i = r_i(n_0) \tag{12}$$

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 ϕ_i, determined, for example, by the contribution the user submitting the task (or his organization) makes to the grid infrastructure, or by the price he is willing to pay for the services he receives, or by any other consideration. We assume, without loss of generality, that the smallest task weight is equal to one. In *weighted* max-min fair sharing, when the demanded rates of the tasks cannot be satisfied, the rate that each task receives is proportional to its weight, provided that the task needs that computation rate.

To be more precise, 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 $\widetilde{N} = \sum_{i=1}^{N} \phi_i$ virtual tasks. An equal fair sharing is performed for all $\widetilde{N}$ virtual tasks using the algorithm of Section 5.1.1. Equations 7, 8, 9 are then modified as follows

$$r_i(n) = \begin{cases} X_i & \text{if } X_i < \phi_i \sum_{k=0}^{n} O(k) \\ \phi_i \sum_{k=0}^{n} O(k) & \text{if } X_i \geq \phi_i \sum_{k=0}^{n} O(k) \end{cases}, \quad n \geq 0 \tag{13}$$

where

$$O(n) = \frac{C - \sum_{i=1}^{N} r_i(n-1)}{card(\widetilde{N}(n))}, \quad n \geq 1 \tag{14}$$

and

$$O(0) = \frac{C}{\widetilde{N}} \quad \text{with} \quad \widetilde{N} = \sum_{i=1}^{N} \phi_i \tag{15}$$

$\widetilde{N}(n)$ 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:

$$\widetilde{N}(n) = \sum_{i} \phi_i : \quad \text{for all} \quad i : X_i > r_i(n-1) \quad \text{and} \quad \widetilde{N}(0) = \widetilde{N} \tag{16}$$

The process is terminated at an iteration n_0 at which either $O(n_0) = 0$ or $card(\widetilde{N}(n_0)) = 0$.

5.2. Fair Task Queue Order Estimation

As already mentioned, 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 where the task is assigned (the *processor assignment problem*). To solve the queue ordering problem in fair scheduling, we will describe shortly, 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. Non-adjusted Fair Completion Time Estimation

We define the *non-adjusted fair completion time* t_i of task T_i as

$$t_i = \delta_i + \frac{w_i}{r_i} \tag{17}$$

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 δ_i (recall that δ_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 δ_i. More precisely, (a) the task is actually assigned to a specific processor j and the earliest starting time on that processor is δ_{ij}, (b) 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), (c) the estimates w_i of the task workloads may be inaccurate. The non-adjusted 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 Simple Fair Task Order (SFTO) rule, the tasks are ordered in the queue in increasing order of their non-adjusted 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 non-adjusted fair completion times are estimated from the non-adjusted 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 6.

5.2.3. Adjusted Fair Task Order (AFTO)

An issue not addressed in the definition of the non-adjusted 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 rates 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, which 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 better approximate the notion of max-min fairness. In the Adjusted Fair Task Order (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 computationally demanding than the SFTO scheme, since the adjusted fair completion times t_i^a are more difficult to obtain than the non-adjusted fair completion times t_i. The way the adjusted fair completion times can be computed is described next. Simulation results on the computation complexity of all schemes will be presented in Section 6.

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 unused processor capacity is divided, it is equally divided among all active tasks. In the second approach, the rates of all active tasks are re-calculated 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 give time [23]. A non-integer 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, i.e., 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

$$F_i(t) = R(\tau) + \frac{w_i}{r_i(t)} \tag{18}$$

where τ 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 τ. $F_i(t)$ 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 Eq. 18, 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,

$$t_i^a : \quad R(t_i^a) = F_i(t_i^a) \tag{19}$$

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 earliest completion time rule (ECT), modified so that it exploits the capacity gaps (Section 4). According to this rule, each task is assigned to the processor that yields the earliest completion time. Simulation results on the performance of the SFTO and AFTO schemes when combined with the ECT rule are described in Section 6.

6. EXPERIMENTAL RESULTS

In this section we describe criteria for measuring the performance of the proposed Grid scheduling algorithms, and present simulation results and comparisons with traditional scheduling policies.

6.1. 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

$$E_1 = \sum_i \frac{\|X_i - X_i^c\|}{X_i} \tag{20}$$

where X_i is the demanded rate and X_i^c is the actual rate allocated to the ith 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

$$E_2 = \frac{\sum_i X_i^c}{C} \tag{21}$$

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 non-preemptable, 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 times,

$$E_3 = \frac{1}{N} \sum_i \frac{\|D_i - max(D_i^c, D_i)\|}{D_i} \tag{22}$$

where D_i is the requested deadline and D_i^c is the actual completion time of the ith 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.

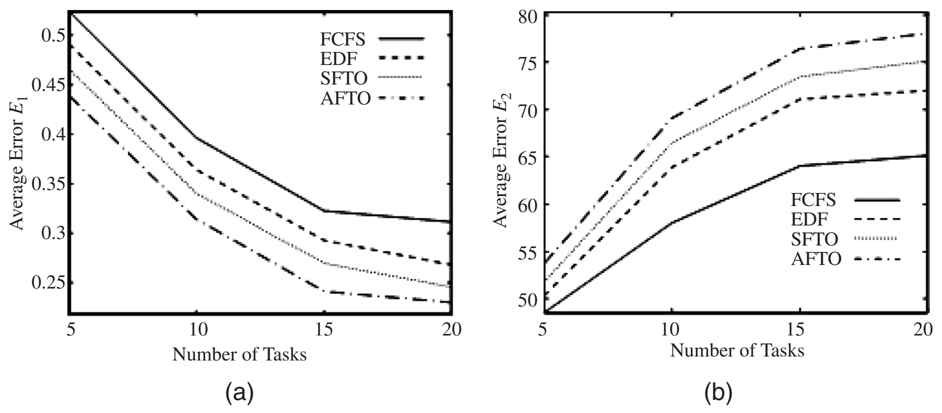
6.2. Simulation Results

The architecture for which the simulation results were obtained consists of 5 processors of different capacities. Several tasks of varying workload, ready times and deadlines are submitted. More specifically, it is assumed that the task workload follows a Gaussian distribution with a varying mean and variance. We define the *normalized load* of the system as

$$\rho = \frac{\sum_i X_i}{C} \tag{23}$$

Considering different values for the variance of the Gaussian distribution, permits the evaluation of scheduling efficiency for the case of symmetric and the asymmetric workloads. Small values of the variance indicate that a number of tasks of similar workload are submitted to the system, while large values indicate very different workloads for the submitted tasks.

Figure 13(a) presents the simulation results obtained for the SFTO and AFTO schemes using criterion E_1, which is plotted against the normalized load ρ. For comparison purposes, we also depict the results obtained for the FCFS and EDF schemes. The simulations were



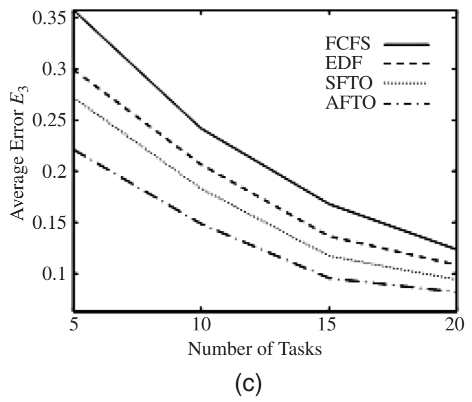
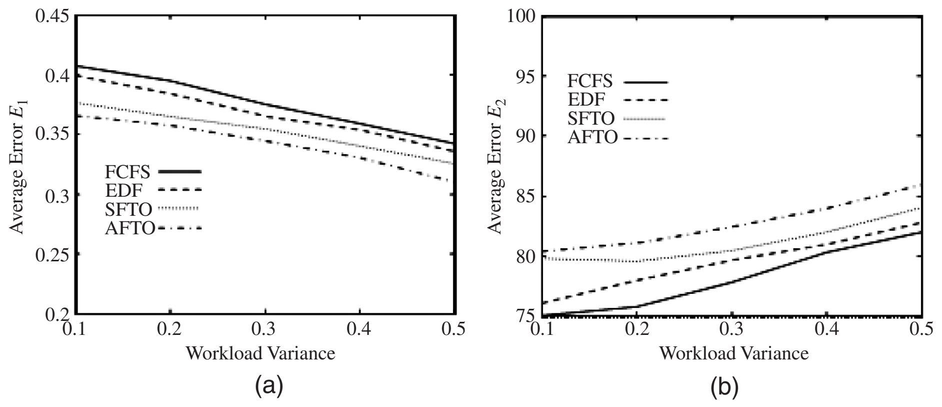


Figure 13. The errors E_1, E_2 and E_3 versus the normalized load ρ for the FCFS, EDF, SFTO, and AFTO policies.



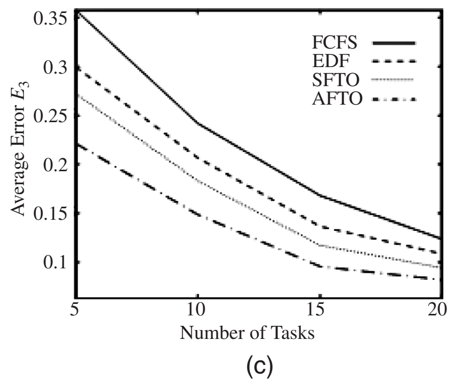


Figure 14. The errors E_1, E_2 and E_3 versus the variance of the workload for the FCFS, EDF, SFTO, and AFTO policies.

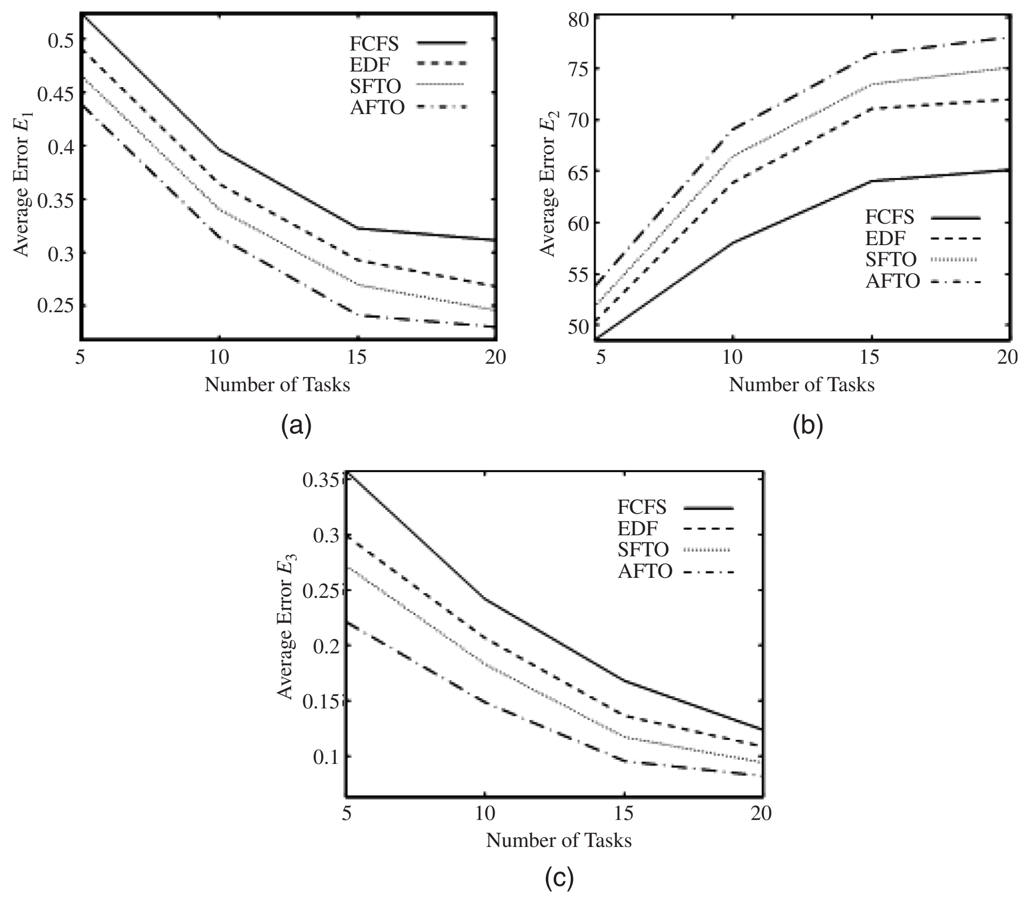


Figure 15. Criteria E_1, E_2 and E_3 versus the number of submitted tasks for the FCFS, EDF, SFTO, and AFTO policies.

performed assuming that the 5 processors have the same computational capacity (symmetric processor case). We observe that the AFTO scheme yields the highest efficiency with respect to criterion E_1. The worst performance is obtained for the FCFS scheme. Figure 13(b) shows the results for the SFTO, AFTO, FCFS and EDF schemes using criterion E_2, assuming similar capacities for all processors. Again, the AFTO scheduling policy yields the best results. Criterion E_3, which expresses the deviation between the demanded deadlines and the actual completion times, is depicted in Fig. 13(c).

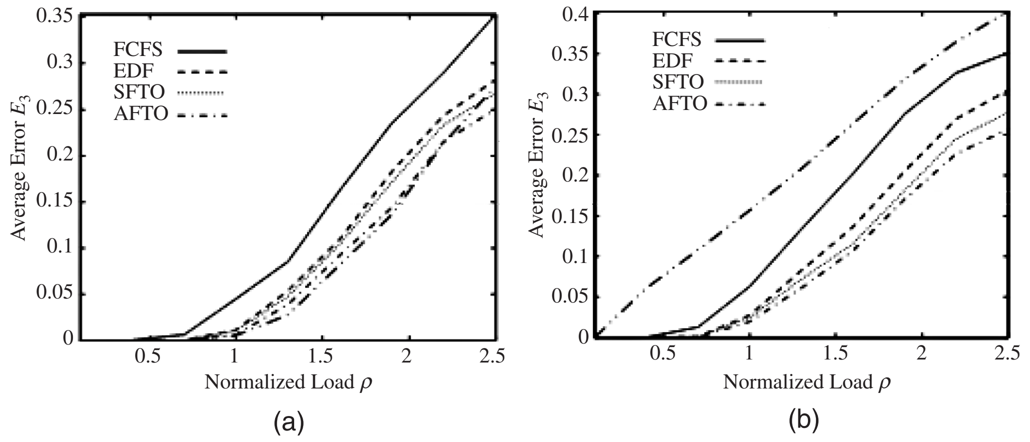


Figure 16. The error E_3 versus the normalized load ρ for the FCFS, EDF, SFTO, and AFTO scheduling policies. (a) Case of medium variation in the processor capacities and (b) Case of very high variation in the processor capacities.

Table 1. Normalized computational complexity for the EDF, SFTO, and AFTO scheduling policies. Normalization has been performed with respect to the complexity of the EDF scheme in case that the number of tasks is 10.

	Normalized	Computational	Complexity		
Number of Tasks	10	15	20	25	30
EDF	1.00	2.00	2.93	3.94	4.87
SFTO	1.01	2.08	3.06	4.15	5.22
AFTO	1.14	2.88	3.94	5.88	15.75

The effect of the workload variance on performance is illustrated in Fig. 14. Particularly, Fig. 14(a–c) plot criteria E_1, E_2 and E_3 for the FCFS, EDF, SFTO, and AFTO policies as a function of the workload variance. The load is held constant to $\rho = 1.25$, the number of tasks is 25, and all processors have equal computation capacities. Note that as the workload variance increases, the performance of all the schemes improves.

Figure 15 presents the effect of the number of tasks to the scheduling performance using criteria E_1, E_2 and E_3 for load $\rho = 1$. It is observed that as the number of tasks increases the scheduling efficiency also increases but with a decreasing rate. This indicates that for a large number of small tasks the scheduling algorithms better exploit the available processor capacity of the Grid infrastructure.

The previous results were obtained assuming small variation in the processor capacities. Figure 16, presents the effect of the variation of the processor capacity (*asymmetric processor case*) with respect to performance criterion E_3. It is observed that the AFTO and SFTO scheduling policies are relatively insensitive to the variation of the processor capacity. For high variation in the processor capacities, the proposed AFTO policy presents the best results, with the SFTO scheme coming second.

The computational complexity of the EDF, SFTO, and AFTO scheduling schemes is presented in Table 1 for different values for the number of tasks requesting service. The complexity has been normalized with respect to the cost of the EDF scheme when the number of submitted tasks is equal to 10. As expected, the complexity for all schemes increases with the number of tasks. The AFTO policy is the most computationally demanding, especially for large number of tasks., because it requires the estimation of the respective adjusted fair rates.

7. CONCLUSIONS

Scheduling efficiency and resource reservations mechanisms are keys to the success of computational Grids, and especially to Grid capability to deliver commercial applications a guaranteed and personalized Quality of Service (QoS). In this paper, we introduce a new resource manager scheme which assigns computational resources to the assigned tasks in an efficient fair way while meeting to the degree possible the quality of service parameters of the individual tasks. For this reason, a new protocol called TARR (Time and Advanced Reservation of Resources) is introduced to estimate the communications delays and the task execution times in order to reserve computational and communication resources only for the time period during which they are actually used by a task and leave those resources available by other tasks for the remaining of the time.

Then, we consider the QoS scheduling problem, in a way that all tasks requesting for service are assigned to the available resources so that the time constraints are satisfied fairly. In this paper, we proposed two new scheduling algorithms for the Grid environment that could be used to implement a fair QoS resource management policy. In the Simple Fair Task Order (SFTO), the tasks are ordered in the queue in an increasing order with respect to their non-adjusted fair completion times. The non-adjusted fair completion times are obtained by the non-adjusted fair computational rates of the tasks by applying a min-max fair sharing algorithm. An improved version of the SFTO scheme is the Adjusted Fair Task order (AFTO) scheduling policy, where the fair rates are dynamically adjusted each time tasks become inactive (e.g., they complete execution) or active (e.g., new arrivals) to better exploit the offered Grid processor capacity. 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 Earliest Completion Time (ECT) policy modified so that processor capacity gaps are taken into account.

Experimental results and comparisons with the traditional First Come Fist Serve (FCFS) and Earliest Deadline First (EDF) scheduling schemes, indicate that our proposed scheduling schemes are fairer and better exploit the available Grid resources. In particular, when the variation in the processors capacities is small, the AFTO scheme outperforms the other schemes with respect to all the examined criteria. The performance of the AFTO scheme remains higher even in case of high variations of the processor capacities. However, the AFTO scheme requires higher computational load than the other scheduling policies.

ACKNOWLEDGMENT

This work was supported by the EU funded project GRILAB.

REFERENCES

1. I. Foster, C. Kesselman and S. Tuecke, “The anatomy of the Grid: Enabling Scalable Virtual Organizations,” International Journal Supercomputer Apllications, 27 (2001).
2. W. Leinberger and V. Kumar, “Information Power Grid: The New Frontier in Parallel Computing,” IEEE Concur., 7, 75–84 (1999).
3. R. Wolski, J. S. Plank, T. Bryan, and J. Brevik, “G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid,” Parallel and Distributed Processing Symposium, 8–11 (2001).
4. Scheduling Working Group Forum, Doc. 10.5 (1999).
5. J.Y-T. Leung and M.L. Merrill, “A Note on Preemptive, Scheduling of Periodic, Real-Time Tasks,” Information Processing Letters, 115–118 (1980).
6. M. S. Fineberg and O. Serlin, “Multiprogramming for Hybrid Computation,” In Proceedings of IFIPS Fall Joint Computer Conference. Washington DC 1967.
7. J. A. Stankovic et al., “Implications of Classical Scheduling Results for Real Time Systems,” Computer, 16–25 (1995).
8. M. L. Dertouzos and A. K.-L. Mok, “Multiprocessor On-line Scheduling for Hard Real Time Tasks,” IEEE Trans. on Software Eng. 1497–1506 (1989).
9. G. Manimaran, C. Siva Ram Murthy, Machiraju Vijay, and K. Ramamritham, “New Algorithms for Resource Reclaiming from Precedence Constrained Tasks in Multiprocessor Real-time Systems,” Journal of Parallel and Distributed Computing, 44, 123–132 (1997).
10. K. Ramamritham, J. A. Stankovic, and P.-F. Shiah, “Efficient Scheduling Algorithms for Real-time Multiprocessor Systems,” IEEE Trans. on Parallel and Distributed Systems, 1, 184–194 (1990).
11. W. Zhao, K. Ramamritham, and J. A. Stankovic, “Scheduling Tasks with Resource Requirements in Hard Real Time Systems,” IEEE Trans. on Software Engineering, 12, 360–369 (1990).
12. X. Deng, N. Gu, T. Brecht, and K.-C. Lu, “Preemptive Scheduling of Parallel Jobs on Multiprocessors,” SIAM Journal on Computing, 30, 145–160 (2000).
13. G. Manimaran and C. Siva Ram Murthy, “An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real-time Systems,” IEEE Trans. Parallel and Distributed Systems, 9, 312–319 (1998).
14. L. E. Jackson and G. N. Rouskas, “Deterministic Preemptive Scheduling of Real Time Tasks,” IEEE Computer, 35, 72–79 (2002).
15. K. Subramoniam, M. Maheswaran, and M. Toulouse, “Towards a Micro-Economic Model for Resource Allocation in Grid Computing Systems,” IEEE Electrical and Computer Engineering, 2, 782–785 (2002).
16. D.P. Spooner, S. A. Jarvis, J. Cao, S. Saini, and G. R. Nudd, “Local Grid Scheduling Techniques Using Performance Prediction,” In IEE Proceedings Computers and Digital Techniques, 150, 87–96 (2003).
17. K. Cooper, A. Dasgupta, K. Kennedy, et. al., “New Grid Scheduling and Rescheduling Methods in the GrADS Project,” IEEE Parallel and Distributed Processing Symposium. 2004, pp. 199–206.
18. Li Keqin, “Experimental Performance Evaluation of Job Scheduling and Processor Allocation Algorithms for Grid Computing on Metacomputers,” IEEE Parallel and Distributed Processing Symposium, (2004), pp. 170–177.
19. 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 Tran. on Networking, 1, 344–357 (1993).
20. A. Demers, S. Keshav and S. Shenker, “Design and Analysis of a Fair Queuing Algorithm,” In Proceedings of the ACM SIGCOMM, (1989), Austin, September.
21. D. Bertsekas and R. Gallager, “Data Networks,” 2nd Edn., Prentice Hall, 1992 (section starting on p.524).
22. 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. on Systems Man and Cybernetics, Part-B, 34, 1235–1247 (2004).
23. S. Keshav, “An Engineering Approach to Computer Networking,” Addison-Wesley, 1997.