# Model of Grid Scheduling Problem

**Pavel Fibich** and **Luděk Matyska** and **Hana Rudová**
Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00, Czech Republic
{xfibich,ludek,hanka}@fi.muni.cz

### Abstract

An extension of Graham's classification of scheduling problems is proposed to cover Grid Scheduling Problems (GSP). The GSP consists of heterogeneous resources in distributed structures (queues), various jobs, and optimality criteria. We will discuss the characteristics (fault tolerance, competitive behavior) of this model and the hierarchies applied for description of the Grid structure.

## Introduction

Efficient use of Grids has become an important problem. A Grid is a structure with distributed heterogeneous resources that are offered to users. Users submit jobs that should be efficiently processed using resources available on the Grid (Nabrzyski, Schopf, & Weglarz 2003). In the Grid Scheduling Problem (GSP) we are looking for an optimal (efficient) assignment of jobs to resources. The resources (e.g., machines, CPUs, memory, storage space) often have a limited capacity and their characteristics change in time (due for example to machine breakdown or consumption of storage space). The Grid environment dynamically evolves as jobs change (due to breakdowns or other interactions) and randomly arrive from users. The goal in this problem is to find an optimal placement of tasks with respect to the costs of the resources assigned. The cost function is often minimized, for example, a maximal estimated completion time of all tasks in the scheduling problem.

The GSP could not be fully classified by Graham's classification of scheduling problems because of the layered structure of the Grid. Scheduling theory (Pinedo 1995; Brucker 1998; Lueng 2004) studies the environments without structures that are suitable for the GSP. We consider an extension of Graham's classification using a hierarchy of queues that reflects the real structure and needs of the GSP. We will also discuss robustness, competitive behavior, the scheduling properties of the Grid structure, and open problems.

This paper is organized as follows. The first section summarizes basic definitions. A scheduling problem is defined by its model (environment, characteristics, constraints and the objective). This model will be described by Graham's classification in the following section. Possible representative of the proposed model will be presented and their properties will be discussed in the end.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

## Basic Definitions

The following definitions are a summary of applicable terms from scheduling theory (Pinedo 1995; Brucker 1998; Lueng 2004) and extensions suited to the GSP (Andrieux *et al.* 2004). Initially, a schedule may be viewed as an assignment of tasks to resources. A more precise definition will be considered after definition of other basic terms. Scheduling is considered to be the activity of creating such a schedule. We will start with a basic classification of resources and job variables, and conclude with a definition of the scheduling problem.

### Resource and Machine

A *resource* is a basic device where jobs are scheduled/processed/assigned (Blazewicz, Brauner, & Finke 2004). Each resource has a limited *capacity* (e.g., number of CPUs, amount of memory). Each resource also has a *speed* and a *load*. Both may evolve in time. The speed defines how quickly the task can be processed on the resource. The load measures how much of the capacity of the resource is used over a time interval.

Now the resources will be classified according to their characteristics. *Processing* resources are needed together with a processor (machine) to complete a given task set. If a resource is needed either before or after processing it is referred to as an *i/o* resource. A resource is *renewable* if its use at any time is constrained, but it may be used again when it is released from a task. The resource is called *nonrenewable*, if its total consumption is constrained (once used by some task, it cannot be assigned to any other task). A resource may be *doubly constrained* if its total usage at any time and the total amount that may be consumed are both constrained. A problem may also contain both *discrete* and *continuous* resources. The classification should also take into account the functions resources fulfill (e.g., CPU, main memory, storage space). Next, resources can be categorized (Baptiste, Pape, & Nuijten 2001) as *disjunctive* (*unary*) or *cumulative*. At most one task can be processed on a disjunctive resource at any time. However, several tasks can

be assigned to a cumulative resource at one time as long as the capacity of the resource is not exceeded.

**Definition 1.** A **machine** (**computing unit**) is a set of cumulative resources (CPUs, memory, storage space, list of specializations) with limited capacities. A machine is described by a name, a set of resources, a list of specializations (e.g., architecture, accelerator), and by its capacity, load, speed and location. All these characteristics are called *descriptors* of the machine. The capacity (speed, load) of the machine can be a general value for the machine or there may be many capacities (speeds, loads), e.g., according to the cumulative resources or the specializations.

## Job

**Definition 2.** A **job** (task, activity) is a basic entity which is scheduled over the resources. A job has specific requirements on the amounts and types of resources (including machines), or required time intervals on these resources, where the job can be scheduled.

The job $j$ has the following variables (properties):

The **processing time** is denoted as $p_j$. If processing time is dependent on the machine ($i$), then we denote it by by $p_{ij}$. If this quantity is not known with certainty in advance, it must be estimated. The processing time may be estimated as the due date of a chosen time queue (often too big), or by using statistics based on recent runs of the chosen program (e.g., Gaussian[1], Amber[2]). Also the processing time is estimated according to the length of the input data or by other advanced techniques (e.g., fuzzy estimate). The estimate may be wrong, though, and the scheduling should watch for this. Defining $p_j$=$p$ means that all jobs have the same processing time $p$. Many algorithms work with this simplification, but we will consider jobs with heterogeneous times.

A **release date** ($r_j$) is the time when a job is available to start processing (including $i/o$ operations). The release date could be explicitly set (a static plan, an allocation in future) or unknown (the release date as the time when the job appears).

A **due date** ($d_j$) is implicitly set by the chosen time queue or explicitly by the user. Both cases are *hard* due dates. The job is killed (ended) after the due date is reached.

A **weight** ($w_j$) signifies the importance of a job $j$, e.g., it may be set according to the user's group (project). It should reflect natural preferences, e.g., a local job has a greater weight than a global one.

A **setup time** ($s_j$) may be used to designate the time required for retrieving (copying) input data or the time for linking to a needed library. This may be dependent on the sequence of jobs. In this case, $s_{ji}$ denotes the setup time needed to process job $j$ after job $i$.

A **machine eligibility restriction** ($M_j$) denotes a set of machines which are capable of processing the job $j$. This may be useful when the user specifies a particular machine architecture or location.

The **start time** ($S_j$) is a time when the job actually begins its processing ($S_j \geq r_j$).

[1]see http://www.gaussian.com/

[2]see http://amber.scripps.edu/

The **completion time** ($C_j$) is the time when a job completes its processing ($p_j = C_j - S_j$ if preemptions are not allowed).

It is important to mention that $d_j$, $r_j$, $C_j$ and $S_j$ are often the wall-clock times. However, $s_j$ and $p_j$ are CPU times. The wall-clock time is often faster or equal to the CPU time (e.g., compare two minutes of the CPU time and two minutes of the wall-clock time).

## Scheduling Problem

Informally we have defined a schedule as an assignment of tasks to resources. A more concrete definition follows.

**Definition 3.** A **schedule of tasks** (or **schedule**) is the assignment of tasks to specific time intervals of resources, such that no two tasks are on any resource at the same time, or such that the capacity of the resource is not exceeded by the tasks.

The schedule of tasks is *optimal* if it minimizes a given optimality criterion (function). The jobs are scheduled to the parts of the machines (e.g., CPUs and memory). Many jobs can be scheduled on one machine at any specific time if the capacities of the resources are not exceeded.

**Definition 4.** A **scheduling problem** is specified by a set of machines, a set of tasks, an optimality criterion, environmental specifications, and by other constraints.

The environment defines relations and connections between the machines and other used structures. This will be described in the $\alpha$ field of Graham's classification. The constraints mainly refer to constraints on resources (e.g., capacity) and jobs (e.g., ordering). A *goal of the scheduling problem* is to find an optimal schedule in the environment and to satisfy all constraints.

We can see that the definition of the scheduling problem is very general. We will begin by concentrating on the optimality criterion, specifications of the environment, and other constraints, and conclude with a more specific definition of the Grid scheduling problem.

# Model of Scheduling Problem

Scheduling problems are often classified according to Graham's classification (Pinedo 1995; Brucker 1998; Lueng 2004; Baptiste, Pape, & Nuijten 2001). This is a three field notation $\alpha \mid \beta \mid \gamma$, where $\alpha$ describes the machine environment, $\beta$ provides details of the processing characteristics and the constraints. The $\gamma$ field contains the objective to be minimized.

## Machine Environment ($\alpha$)

The basic notations of machine environments are *single machine* ($1$), *identical machines in parallel* ($Pm$), *machines in parallel with different speeds* ($Qm$), *unrelated machines in parallel* ($Rm$), and *(flow|open|job) shop* ($Fm$, $Om$, $Jm$).

When $1$ is in the $\alpha$ field, it indicates that we are working with only one machine where jobs can be scheduled. A single machine can be seen as a disjunctive resource and other environments as cumulative resources. We could schedule the jobs on $m$ identical machines in a parallel environment

$Pm$. This environment is widely discussed (Pinedo 1995; Lueng 2004) and has many feasible methods for solving. If the machines in $Pm$ instead have different speeds, then the environment is denoted as $Qm$. $Rm$ is a generalization of $Qm$ where different speeds of the machine are used for different jobs.

It is important to mention that machines used here are not the same as in Definition 1. A machine in Graham's classification corresponds to a unary resource (processor), but we consider (in Definition 1) that a machine has a set of cumulative resources (e.g., processors, memory).

The shop environments apply the idea that the task is processed on multiple machines. Each task must be processed on every machine in the same order in $Fm$. In $Om$, each task must be processed on every machine in an order that is not known in advance and scheduling must choose the order for all tasks. $Jm$ is close to $Om$, but differs in that the order in which tasks are processed is known in advance.

## Grid Hierarchy ($\alpha$)

We suppose a more structured environment in the Grid scheduling problem (Nabrzyski, Schopf, & Weglarz 2003) than has been defined for the previous environments. An extension of the machine environment field in Graham's classification is therefore suggested. The actual structure of a Grid can be seen as a hierarchy that includes queues, machines and other resources. Our motivation here is the real-life Grid structure applied in the $\mathcal{META}$ *Centrum*[3], a national computing and data Grid operated by $CESNET$, and having its nodes distributed at the Supercomputing centers at Masaryk University in Brno, Charles University in Prague, and University of West Bohemia in Pilsen. The hierarchy should provide easy access to the Grid facilities and scheduling capabilities.

Let us first consider the term queue and its types.

**Queue** A *queue* is an abstract structure that accepts, schedules, and/or sends jobs that have been submitted by users to resources that belong to the queue. The queue often has a set of jobs it has accepted that are waiting for scheduling. It may also contain a set of scheduled jobs. Either these copies of jobs previously sent to the resource(s) are saved by the queue in case there is a failure of a resource or these copies wait until the chosen resource(s) will be free. The queue $Q_i$ has a set of resources (machines) that we denote $Q_i.R = M_{i1} \cup \ldots \cup M_{im_i}$, where $M_{ij}$ is $j$-th machine that belongs to the queue $Q_i$. The machines in the $Q_i.R$ define a restriction on the set of eligible machines ($M_j$). The length of the queue indicates the load of resources that belong to the queue or/and the number of jobs waiting for processing or scheduling in this queue. Here we assume that a resource may be either a machine or another queue.

Now consider three basic types of queues that are included in the hierarchies.

A *global queue* $GQ$ schedules jobs to resources that belong to it and/or sends the jobs to a local queue ($LQ$) belonging to the $GQ$. The global queue is a main entry point to the hierarchy because it can distribute jobs among many resources in the hierarchy. The parameters of the $GQ$ indicate which local queues ($LQs$) may be used for scheduling a job entering the hierarchy via the $GQ$.

A *local queue* $LQ$ schedules jobs to resources that belong to it and/or sends the jobs to a FIFO queue $FQ(s)$ belonging to the $LQ$. The local queue is an alternative entry point to the hierarchy and it often distributes jobs among a smaller set of resources than those available to the $GQ$. The $LQs$ have parameters that indicate:

- the length of time a job may be processed (run) on a resource belonging to the $LQ$,
- the limit on the number of users (jobs) that can be in the $FQs$ belonging to the $LQ$,
- the limit on the number of machines that can be assigned to one job and other limits.

The motivation for these parameters is the $Portable\ Batch\ System$ ($PBS$) in the $\mathcal{META}$ *Centrum*[4]. The local queues decompose the scheduling decision of the global queue, so as to be more easily computed.

A *FIFO queue* $FQ$ accepts the jobs from some $LQ$ and such jobs are processed (run) on the resources (machines) that belong to this queue, nowhere else. The $FQ$ does not make any scheduling decisions. It applies a First-In First-Out method to run the jobs on the free resource(s) or on the resource(s) that was/were explicitly defined in the $GQ$ or $LQ$, or by the user who entered the job. If the required resource is not free, the job has to wait in the queue and it blocks similar jobs that were also accepted by the $FQ$ and that require the same resource. If jobs with a high weight may forerun jobs with a lower weight, then we speak about a weighted FIFO queue. Each FIFO queue can serve one or many machines.

A queue schedules jobs on the resources that belong to it. The $FQ_i$ schedules jobs on $FQ_i.R = M_{i1} \cup \ldots \cup M_{im_i}$. The $LQ_j$ schedules jobs on the resources of all $FQ_i$ that belong to $LQ_j$, $LQ_j.R = \cup_i FQ_i.R = \cup_{i,k \leq m_i} M_{ik}$. Likewise, the $GQ_k$ schedules jobs on the resources of all $LQ_j$ that belong to $GQ_k$, $GQ_k.R = \cup_j LQ_j.R$.

**Queue Copies** The queue may be copied into new queue(s). Such new copies have the same properties as the original queue. The resources of the original queue are shared between the original and the copied queues. This method is often used to accommodate a possible unavailability of the original queue. This will be discussed in greater detail later. Two methods of communication between the original and copied queues may be applied, each resulting in different behavior. The queue and its copies may either be synchronized or only one may be active at any moment.

When the queues are synchronized, all queues are active, accept jobs, and have the same schedules. However, only one of the queues sends jobs to other queue(s) or re-

[3]Informations about the $\mathcal{META}$ *Centrum* are available in http://meta.cesnet.cz/cms/opencms/en/

[4]More about the $PBS$ in the $\mathcal{META}$ *Centrum* is available in http://meta.cesnet.cz/cms/opencms/en/docs/software/system/pbs.html

source(s). In the second method, only one queue is active and accepts (schedules) jobs. The remaining copies of the queue are inactive and only wait in case the active queue crashes (becomes unavailable). One of them then substitutes for the crashed queue. Both methods require some synchronization. The first method synchronizes the schedule and the second synchronizes the active state. For the original queue $Q_i$, we denote by $Q_j^{si}$, the queue that is *synchronized* with the $Q_i$. And we denote by $Q_j^{ni}$, the queue that is a *non-active* copy of the $Q_i$.

Only global and local queues can be copied. The FIFO queues are tightly coupled with the resources and there is no reason to duplicate them. The copy of some queue must have the same type (global, local) as the origin one.

**Competitive Behavior** Competitive behavior for the resources of queue $Q_i$ refers to the situation where the capacity of the resources in $Q_i$ is divided among other queues. A percentile of the resources use (or simply usage of the queue) is often applied here. This percentage is computed over a chosen time interval and small deviations are allowed. For example, when two queues $Q_j$ and $Q_k$ both want to use $Q_i$, we might indicate that the usage of $Q_i$ is divided 30 % for $Q_j$ and 70 % for $Q_k$. Generally, we want to denote competitive behavior for $Q_i$ among the queues $Q_1, \ldots, Q_j$ in $pr_1, \ldots, pr_j$ percentages, where $Q_l$ should get $pr_l$ percent of the resources in $Q_i$ and $pr_1 + \ldots + pr_j = 100$ %. This is denoted as $j$ new queues $Q_i^{pr_1}, \ldots, Q_i^{pr_j}$ where the $Q_i^{pr_k}$ belongs to $Q_k$.

**Hierarchy** We now denote a hierarchy of queues for further use in Graham's three field classification. Let

$$\begin{aligned}\mathcal{H}_3 = GQ_1\{&LQ_{1,1}, \ldots, LQ_{1,l1}\}[FQ_{1,1}, \ldots, FQ_{1,f1}],\\ &\ldots,\\ &GQ_k\{LQ_{k,1}, \ldots, LQ_{k,lk}\}[FQ_{k,1}, \ldots, FQ_{k,fk}]\end{aligned}$$

be a basic 3-layer hierarchy with $k$ global queues with underlying local and FIFO queues. The brackets are applied to describe which queues belong to other queues. It may also happen that some global queues share same FIFO queues. Notation for two such global queues is

$$\begin{aligned}&(GQ_{i_j}\{LQ_{i,1}, \ldots, LQ_{i,li}\},\\ &GQ_j\{LQ_{j,1}, \ldots, LQ_{j,lj}\})[FQ_{ij,1}, \ldots, FQ_{ij,fij}] \ .\end{aligned}$$

It means that $GQ_i$ and $GQ_j$ have still different local queues but they share same FIFO queues. The *cp_co*_$\mathcal{H}_3$ hierarchy is an extension of the previous hierarchy. The *cp* parameter denotes that some copies of the global or local queues are used. The *co* parameter notates competitive behavior for some FIFO queues. The $\mathcal{H}_2$ defines the hierarchy without the local queues or global queues. We say that the hierarchy is *centralized* if the jobs are scheduled in only one place (often in the $GQ$). The opposite is a *decentralized* hierarchy where scheduling decisions are decomposed among parts of the hierarchy (decisions are made in the global and local queues, or in copies of the queues).

For example, the decentralized hierarchy in Figure 1 may be denoted as

$$\begin{aligned}cp_co_\mathcal{H}_3 = (&GQ_1\{LQ_1, LQ_2\}, GQ_2^{n1}\{LQ_3^{s1}, LQ_4^{s2}\})\\ &[FQ_1, FQ_2, FQ_3^{30}, FQ_4], GQ_3\{LQ_5,\\ &LQ_6, LQ_7\}[FQ_5, FQ_6, FQ_3^{70}, FQ_7].\end{aligned}$$

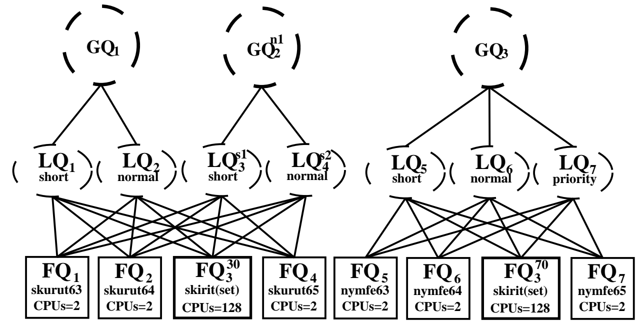


Figure 1: A decentralized *cp_co*_$\mathcal{H}_3$ hierarchy.

In this example, there is an inactive copy $GQ_2^{n1}$ of the $GQ_1$, synchronized copies $LQ_3^{s1}$ and $LQ_4^{s2}$ of $LQ_1$ and $LQ_2$, and competition for FIFO queue $FQ_3$. The $GQ_2^{n1}$ and $GQ_1$ consist of different local queues but they share the FIFO queues. The parameters listed under the local queues define their types. The name of the machine(s) that belong to the $FQ$, along with the number of CPUs is listed under the $FQ$.

## Characteristics and Constraints ($\beta$)

We want to look first at the two parameters relating to tasks and then discuss machine unavailability types and resource constraints. The main types of online scheduling are also mentioned.

Preemptions ($prmp$) allow us to change the resource on which a task runs. These must be considered with respect to other parameters (e.g., restart of a task after preemption, whether the task must continue on the same resource after the preemption, and the possibility of checkpoint-ability of the job).

Precedence constraints ($prec$) specify an order in which the jobs can be processed (e.g., first retrieve the data and then compute on them). They are often in a $tree$ format, where the schedule must start with the job in the root or with some jobs in the leaves. A $chain$ format can also be considered, where jobs are ordered in chains. The $prec$ parameter is also called a *workflow*.

**Machine Unavailability (Breakdown)** A machine may be unavailable due to maintenance or a breakdown. There are three different cases of working with unavailable machines (Lee 2004).

**Resumable (*r-a*):** The machine is $resumable$ if the unfinished job can continue after the end of the unavailable interval without any penalty.

**Non-resumable (*nr-a*):** The machine is *non-resumable* if the job must be fully restarted after the unavailable interval.

**Semi-resumable (*sr-a*):** The machine is *semi-resumable* if the job must be partially restarted after the unavailable interval.

**Online Scheduling** Online scheduling supposes that jobs arrive over time or one by one (Pruhs, Sgall, & Torng 2004; Sgall 1997). Jobs are scheduled without any knowledge of the future, and often without knowledge of the processing

times of jobs. An online schedule is compared with an optimal schedule by competitive or worst-case analysis. Three different types could be in the $\beta$ field.

In **online-time**, scheduling must decide which job has to run at time $t$. Once a job is released, we assume that the scheduler learns the processing time of a job (e.g., a web server serving static documents).

In **online-time-nclv**, the scheduler is given no information about the processing times (e.g., in the operating system). This lack is called non-clairvoyance.

In **online-list**, the jobs are ordered in a list/sequence. As soon as the job is presented, we know all its characteristics. The job is then assigned to some machine and time slots (consistent with restrictions). The scheduler cannot change this assignment once it has been made (e.g., load balancer working in front of a server farm).

### Objective Function ($\gamma$)

Here we consider the objective functions that evaluate the quality of solutions. There are two groups of scheduling objective functions.

In the first group there are basic functions.

- A **makespan** ($C_{max}$) defined as $max(C_1, \ldots, C_n)$, the completion time of the last job. This objective function formalizes the viewpoint of the owner of the machines. If the makespan is small, the utilization of the machines is high.
- A **total weighted completion time** ($\sum w_j C_j$) or the total unweighted completion time defined as $\sum C_j$. This objective is more suitable for the user, the time it takes to finish individual jobs may be more important.
- A **maximum lateness** ($L_{max}$) defined as $max(L_1, \ldots, L_n)$, where $L_j = C_j - d_j$ (the difference between the completion time and the due date).
- A **total weighted tardiness** ($\sum w_j T_j$) or the unweighted case ($\sum T_j$), where $T_j = max(L_j, 0)$ and $L_j = C_j - d_j$.
- A **weighted number of tardy jobs** ($\sum w_j U_j$) or the unweighted case ($\sum U_j$), where $U_j = 1$ if $C_j > d_j$; otherwise $U_j = 0$.

The second group consists of objective functions that are more suitable for online scheduling. They define the *flow* of a job as $F_j = C_j - r_j$ (the difference between the completion time and the release date) and the *stretch* of a job as $St_j = (C_j - r_j)/p_j$ (the flow of the job over the processing time).

- A **total weighted flow time** ($\sum w_j F_j$) or the unweighted case ($\sum F_j$).
- A **total weighted stretch** ($\sum w_j St_j$) or the unweighted case ($\sum St_j$).
- A **maximum flow time** ($max\ F_j$).
- A **maximum stretch** ($max\ St_j$).
- A $l_p$ **norm of the flow times** ($(\sum {F_j}^p)^{1/p}$), provides a QoS measure for a collection of jobs.

The previous functions may be combined together to allow optimization over more than one function.

## Grid Scheduling Problem

Our discussion of the parameters of Graham's classification can be concluded by applying them in the following definition.

**Definition 5.** The **Grid scheduling problem** (**GSP**) is the scheduling problem defined on

$$Rm, cp_co_\mathcal{H}_3 \mid r_j, d_j, s_{jk}, M_j, nr-a, online-time \mid f_o$$

We now consider the purpose of the resources and parameters used in the definition of the GSP.

**Resources** We will work with processing resources (processors, memory) and i/o resources (storage space). The i/o resources allow us to retrieve data at the beginning of the task and to save the data at the end of the task. The input resource for the job $j$ could be modeled as the setup time $s_{0j}$, where the jobs are numbered from 1. And similarly for the output resource $s_{j0}$. All other resources may be supposed to be processing resources.

Renewable resources will also be considered in our case (machine, processor, memory) along with the doubly constrained category (storage space).

Discrete resources are supposed only, because we do not need a continuous division of resources.

A CPU is a disjunctive resource, but in the definition of our machine we work with a set of CPUs, which is a cumulative resource. The memory and storage space are also cumulative resources.

**$\alpha$ field** We concentrate on $Rm$ because a Grid consists of unrelated machines with different speeds for different jobs. This could describe the specialized architectures (e.g., for matrix or graphic operations). However, there are more solution methods for the machines $Qm$ and $Pm$ and those can be generalized to $Rm$. Here it is important to mention that we will work with machines in the sense of Definition 1.

Shop environments are more closely related to manufacturing problems (producing similar products) than to the Grid scheduling problem (heterogeneous jobs).

The basic information about the hierarchy $cp_co_\mathcal{H}_3$ has been discussed above and further discussion about the characteristics of a suitable hierarchy is deferred until the end of this paper.

**$\beta$ field** A list of the variables and the constraints is included in the $\beta$ field of the GSP. We consider the following parameters.

The $r_j$ parameter defines that jobs have specified release dates. It may be useful to an allocation in the future. Most jobs want to start processing when they are accepted by some queue. This parameter is required for time allocation.

The $d_j$ parameter indicates that jobs have specified due dates. This is a significant parameter of the job, because violation of the due date implies killing a job.

The $s_{jk}$ is used when jobs have specified sequence dependent setup times. We will include this parameter in the modeling of i/o resources.

The $M_j$ means that jobs may be processed only on a subset of all machines. This parameter assists with specification of special architectures.

The **preemption** (*$prmp$*) of the job could bring a high cost (e.g., repeating transport of input data). We will not work with this parameter initially as it may be more complicated for modeling.

The **precedence constraints** (*$prec$*) may be included when the user can partition a large job into smaller jobs and specifies the order that is needed to accomplish all jobs. Initially, we will not consider this feature not to have a need to precise workflows.

The **breakdowns** (*$brkdwn$*) parameter indicates that the machines (resources) can break down. More accurate are *r-a*, *nr-a* and *sr-a* parameters (see section on breakdowns). We are interested in the *nr-a* case, at the beginning, and checkpoint-ability of a job will be added later.

The *online-time* parameter defines that a job that appears is scheduled as early as possible, and we have estimated its processing time (our case). For the *online-time-nclv* type a round-robin, or other preemptive balancing methods often proposed. The *online-list* is close to batch scheduling, where the time between assignments of jobs to the machine is meaningless (for online types see section about online scheduling).

**$\gamma$ field ($f_o$)** We will work with weighted jobs. This implies working with weighted objective functions. We will also include the due dates and the release dates (implicitly by the *online-time* parameter). The suitability of a maximum or a total objective is compared by including the weights and the dates. The maximum is not as strong (but has lower computationally cost) than the total objective, which includes the variables of all jobs. The objectives $\sum w_j U_j$ and $\sum w_j T_j$ are the most interesting basic functions without the online assumption. With the online assumption, the objectives $\sum w_j F_j$ and $\sum w_j St_j$ are more favorable. We are more interested in the due dates than the release dates, so the first two objectives would be better.

We may also use a multi-criterion objective function that can be defined to optimize several objective functions.

## Discussion about Hierarchies

We now consider various hierarchies of queues and discuss their characteristics to find one best suited to our needs.

**Example 1 (trivial structure).** Let us start with the example in Figure 2 where the hierarchy

$$\mathcal{H}_2 = GQ[FQ_1, FQ_2, FQ_3, FQ_4]$$

without the local queues is depicted. Jobs are submitted by users to the pool of jobs to be scheduled in the global queue $GQ$. The $GQ$ knows the parameters of all jobs in the pool. It chooses a free $FQ$ and a job for scheduling. The $FQs$ send the $GQ$ information about their loads. If a $FQ$ announces that it is free, then the $GQ$ chooses one job from the pool for the free $FQ$ to process. The $GQ$ then sends the chosen job to the $FQ$ for processing. The $GQ$ has to do some preprocessing for the scheduling decision. In this hierarchy, the $FQ$ may be a machine.

This hierarchy is feasible for homogeneous jobs. If we consider heterogeneous jobs, though, there may be a starvation of hard jobs processed in the queue. If only one machine is free, an easy job requiring few resources will be more feasible to schedule for processing. A hard job, requiring many resources to be available at one time, likely must wait until there are no easy jobs and sufficient resource capacity is available. A similar structure is applied in the $Condor$[5] project, where a match-making between the requirements of the jobs and the abilities (e.g., number of CPUs) of the $FQs$ is evaluated.

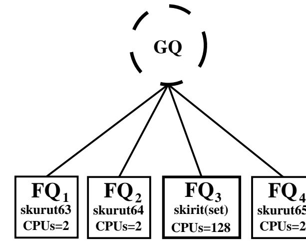


Figure 2: An example of the centralized $\mathcal{H}_2$ hierarchy with one Global queue ($GQ$) and the FIFO queues ($FQ$).

The $GQ$ is a bottleneck in this hierarchy for two reasons.

1. If the $GQ$ is unavailable then no jobs (new and from the pool) can be scheduled. This will be discussed in the following section.
2. It is difficult to schedule many jobs on many resources in only one place.

**Fault Tolerance** We will discuss a problem if failures (crashes) of some parts of the hierarchy happened. The hierarchy is *fault tolerant* if for all parts of the hierarchy it holds that if a failure of some part happens, there is another part that substitutes for the function of the failed part. The function of the part (e.g., queue, machine) may be accepting, processing, and/or scheduling of jobs. We are interested in failures of machines and queues. We do not consider crashing jobs, as it is difficult to automatically find the reason of the failure (it can be a wrong scheduling, the misconfiguration of a machine, or user's mistake).

Fault tolerance of the global and local queues can be achieved by copying the queues in such a way that there is some queue that accepts and schedules jobs at every time. We do not make a comparison of the methods of communication between copies of the queues. However, we note that the task of synchronization of the schedule among queues becomes harder. The FIFO queues and machines cannot be copied. We must save their state in the higher levels of the hierarchy. The *state* of some resource is the schedule for this resource and information about jobs in the schedule which could be re-scheduled. This idea may be applied on the whole hierarchy, such that each part of the hierarchy saves copies of the states of the lower parts. However, we should be watchful for too many copies of the states and for the cost of saving the state. For example, when some $FQ$ has a failure, its state is saved in some $LQ$ and $GQ$. And the re-scheduling of jobs that were in the $FQ$ could be done by the $LQ$ or $GQ$.

[5]Details about $Condor$ are in http://www.cs.wisc.edu/condor/

In Example 2, we could solve the first reason for the bottleneck of the $GQ$ by copying the $GQ$ and by copying the state of all $FQs$ into all global queues. This solution is shown in Figure 3 where the hierarchy

$$cp_\mathcal{H}_2 = (GQ_1, GQ_2^{s1})[FQ_1, FQ_2, FQ_3, FQ_4]$$

with two global queues that share the same FIFO queues is described.

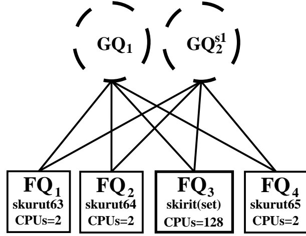


Figure 3: An extended example, the $cp_\mathcal{H}_2$ hierarchy.

**Example 2 (decomposition).** We now focus on the decomposition of the $GQ$ from the previous example. In the distributed environment, it is often suitable to decompose a hard problem into sub-problems. Therefore, we decompose the $GQ$ into smaller parts where it should be easier to compute a schedule.

Look at the example in Figure 4 where the decentralized hierarchy

$$\mathcal{H}_3 = GQ_1\{LQ_1, LQ_2, LQ_3\}[FQ_1, FQ_2, FQ_3, FQ_4]$$

is shown. The global $GQ$ was decomposed into the $GQ_1$

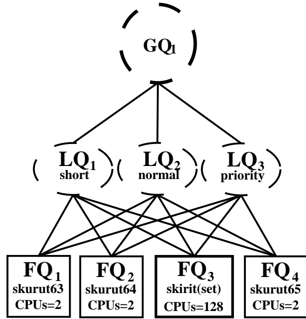


Figure 4: An example of the decomposition of the global queue $GQ$ and the $\mathcal{H}_3$ hierarchy.

and $LQ_1, LQ_2, LQ_3$. The jobs should enter the hierarchy by some global queue (in Figure 4 by the $GQ_1$). However, the job could specify some $LQ$, $FQ$ or directly some machine. The parameters under local queues indicate the groups where the $GQ_1$ schedules (divides) the jobs. Here, we use the time of processing and the priority classes of the jobs. Short jobs are scheduled in $LQ_1$, priority jobs in the $LQ_3$ and other jobs are scheduled in $LQ_2$. Choosing a suitable $LQ$ is often the first part of scheduling. The $LQ$ then has to choose some suitable $FQ$.

Decomposition of the scheduling decision may be more suitable for decentralized structures like the Grid modeled by the hierarchy in Figure 4. We should define some uniform distribution of the jobs between the local queues that reflects real world usage. Such distributions often divide jobs according to the estimated time of processing, their priority, resources specified, locations, etc.

The fault tolerant solution of the hierarchy in Figure 4 is the same as the solution for the hierarchy in Figure 2. We make copies of the decomposed queues (as the old $GQ$) and save all states. This solution is illustrated in Figure 5 where the hierarchy

$$cp_\mathcal{H}_3 = (GQ_1, GQ_2^{s1})\{LQ_1, LQ_2, LQ_3, LQ_4^{s1}, LQ_5^{s2}, LQ_6^{s3}\}[FQ_1, FQ_2, FQ_3, FQ_4]$$

is depicted. Here, the synchronized copy $GQ_2^{s1}$ of the $GQ_1$

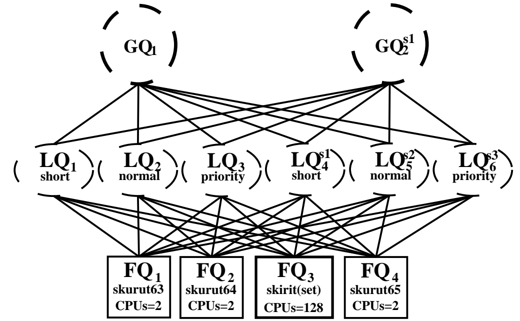


Figure 5: An example of the fault tolerant $cp_\mathcal{H}_3$ hierarchy.

forms a second entry point to the hierarchy. Both global queues share the same local queues. The local queues are also copied and synchronized in case of a possible failure in some of them. For the same reason, the global queues can send jobs to the original and copied queues.

**Example 3 (competitiveness).** Competitive behavior for the queue $Q_i$ is a situation where the capacity of the resources of $Q_i$ is divided between the $Qs$. We may say that the $Qs$ share the resources of $Q_i$.

Look at the example in Figure 6 for the hierarchy

$$co_\mathcal{H}_3 = GQ_1\{LQ_1, LQ_2, LQ_3\}[FQ_1, FQ_2, FQ_3^{30}, FQ_4], GQ_2\{LQ_4, LQ_5, LQ_6\}[FQ_5, FQ_6, FQ_3^{70}, FQ_7]$$

where there are two independent global queues with similar structures that belong to the hierarchy. However, the same shared FIFO queue $FQ_3, skirit(set), CPU = 128$ is included in both global queues. The $GQ_1$ can use 30 % of the $FQ_3$ capacity and the $GQ_2$ can use 70 %. This real requirement, the competitive behavior, has to be handled somehow. There is only one $FQ_3$. $FQ_3^{30}$ and $FQ_3^{70}$ are *virtual* queues. We must send the jobs from the virtual queues to the single real $FQ_3$ that can process the jobs.

Here we must also solve how and where to compute the required percentages of use. These percentages should be computed in the original shared resource (e.g., $FQ_3$ in Figure 6), but may also be computed in the $LQs$ or $GQs$.

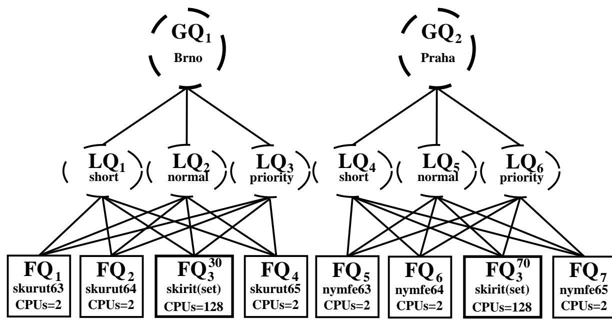


Figure 6: An example of the $co_\mathcal{H}_3$ hierarchy.

The local queues to which the virtual queues belong should be informed of the actual percentages of used capacity. This is suitable for the situation when a large deviation from the percentages occurs. Then, such local queues must not send jobs to the shared resource until the percentages are repaired.

**Suitable Hierarchy** In the previous examples, we discussed the characteristics of hierarchies and tried to highlight their problems. Decomposition of the global queue into a new global queue and smaller local queues is more suitable for decentralization and load balancing of the hierarchy. This decomposes scheduling into small independent parts and can handle situations when some parts are unavailable. Using additional copies of the global and local queues helps us in situations where there are failures to queues.

Saving the states of resources can also solve the problem of failures of these resources. Also, in case of significant changes to the load on the resources, or diversion of the capacity of the shared resources, we can easily re-schedule. This is because the states have been saved in the higher levels of the hierarchy.

In Grid environment, we have a small number of global queues either serving groups of users, or as copies for fault tolerance. The groups are often divided by location, institution, and/or by specialization. The different global queues often use different resources, but they may share or make competition for them.

The connections between the global and local queues, and between local and FIFO queues are often evolving. In our examples, each local queue can send a job to each FIFO queue that belongs to the common global queue. However, this situation is not true for all instances, because such connections may be added or removed for other reasons (e.g., network crashes) or by scheduling policy (e.g., deviation in percentages of use in competitive behavior).

The distribution of the local queues should reflect the distribution of job requirements. Independent groups of machines or an independent single machine should form a FIFO queue.

## Conclusion and Future Work

In this paper, we have defined the basic terms and summarized the elements of the Grid scheduling problem (GSP). Resources and job properties have been discussed. We have suggested a model for the scheduling problem based on an extension of Graham's classification. A Grid hierarchy, such as a machine environment, was designed. This consists of a few types of queues, that are applied for scheduling, and of machines. We presented a concrete definition of the GSP. The discussion about hierarchies summarized the properties and problems of the hierarchies. We focused on fault tolerance and competitive behavior. We suggested solving methods for the first property by copying and decomposing the queues and by saving the state of the resources. A solution for competitive behavior was also proposed. The discussion concluded with suggestions for a hierarchy that is suitable for the GSP.

Future work will be focused on formalizing the characteristics of the hierarchies and discussion of them. We will extend the model for preemption and precedence constraints. Our work on modeling should conclude with choosing a suitable Grid hierarchy with precise definitions of its characteristics and functions. The modeling of the problem introduces the first step in the proposal of suitable solving methods for scheduling tasks in the Grid environment. We would like to extend the model in direction towards the constraint optimization model and apply constraint programming (Dechter 2003) to solve the extended GSP.

## References

Andrieux, A.; Berry, D.; Garibaldi, J.; Jarvis, S.; MacLaren, J.; Ouelhadj, D.; and Snelling, D. 2004. *Open issues in Grid scheduling*. Technical report, National e-Science Centre and the Inter-disciplinary Scheduling Network.

Baptiste, P.; Pape, C. L.; and Nuijten, W. 2001. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers.

Blazewicz, J.; Brauner, N.; and Finke, G. 2004. *Scheduling with Discrete Resource Constraints*, In Lueng (2004). chapter 23.

Brucker, P. 1998. *Scheduling Algorithms*. Springer.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Lee, C. Y. 2004. *Machine Scheduling with Availability Constraints*, In Lueng (2004). chapter 22.

Lueng, J. Y.-T., ed. 2004. *Handbook of Scheduling: Algorithms, Models and Performance Analysis*. CRC Press.

Nabrzyski, J.; Schopf, J. M.; and Weglarz, J. 2003. *Grid Resource Management: State of the Art and Future Trends*. Kluwer Publishing.

Pinedo, M. 1995. *Scheduling : theory, algorithms and systems*. Prentice Hall.

Pruhs, K.; Sgall, J.; and Torng, E. 2004. *Online Scheduling*, In Lueng (2004). chapter 15.

Sgall, J. 1997. Online scheduling – a survey. In Fiat, A., and Woeginger, G., eds., *On-Line Algorithms*, Lecture Notes in Computer Science. Springer-Verlag, Berlin.