J Sched (2013) 16:349–368
DOI 10.1007/s10951-011-0254-9

Hierarchical scheduling strategies for parallel tasks and advance reservations in grids

Krzysztof Kurowski · Ariel Oleksiak · Wojciech Piątek · Jan Węglarz

Abstract Recently, the advance reservation functionality gained high importance in grids due to increasing popularity of modern applications that require interactive tasks, co-allocation of multiple resources, and performance guarantees. However, simultaneous scheduling, both advance reservations and batch tasks affects the performance. Advance reservations significantly deteriorate flow time of batch tasks and the overall resource utilization, especially in hierarchical scheduling structures. This is a consequence of unknown batch task processing times and the lack of possibility of altering allocations of advance reservations. To address these issues we present a common model for scheduling both computational batch tasks and tasks with advance reservation requests. We propose simple on-line scheduling policies and generic advices that reduce negative impact of advance reservations on a schedule quality. We also propose novel data structures and algorithms for efficient scheduling of advance reservations. A comprehensive experimental analysis is presented to show the influence of advance reservations on resource utilization, mean flow time, and mean tardiness—the criteria significant for administrators, users submitting batch tasks, and users requesting advance reservations, respectively. All experiments were performed with a well-known real workload using the GSSIM simulator.

Keywords Grid computing · Grid resource management and scheduling · Scheduling with advance reservation · Grid simulation · Real workloads

1 Introduction

New-generation supercomputers available in grid environments achieved petaflop-level performance and they allow end-users to execute highly demanding applications with interactive visualization as well as real-time data exchange between computing resources over high-speed networks. Many researchers have focused their efforts on various parallel task batch scheduling techniques, often without using advance reservation capabilities. In general, there were three reasons why advance reservation techniques were not widely adopted in production grids. The first one was related mostly to technical issues as various *local resource management systems (LRMSs)* did not support advance reservation features, supported them partially, or those features were not exposed to upper-level grid middleware services and protocols. The second reason was related to many empirical tests demonstrating that using advance reservation techniques often results in low resource utilization rate, high rejection rate and longer response time. The third problem was mostly reported by local administrators and related to difficulties with an accurate estimation of task runtimes and advance reservation parameters provided by end-users. Our analyses of many different real workloads collected over the last decade also confirmed that end-users often do not specify task runtimes during the task submission process. However,

---

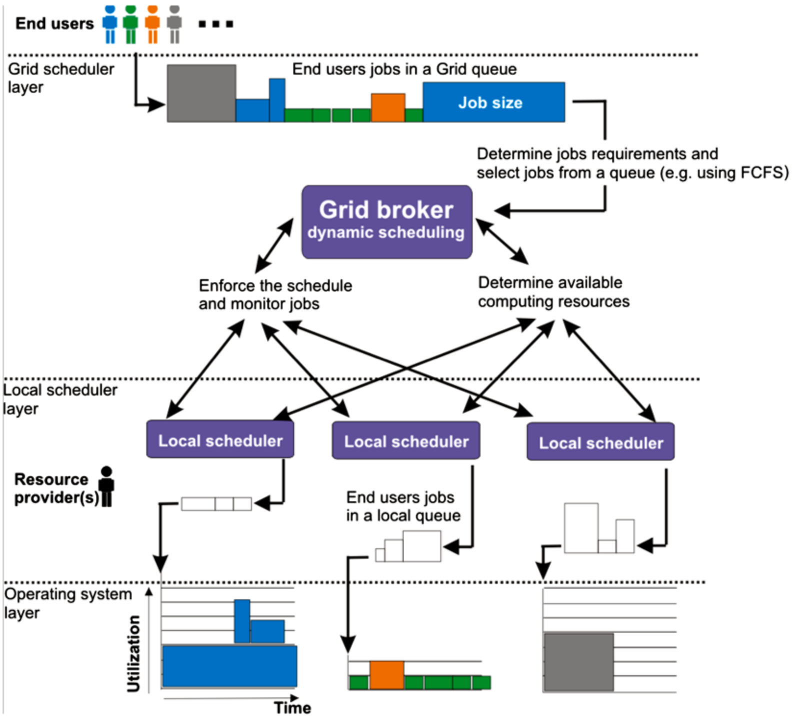
K. Kurowski · A. Oleksiak (✉) · W. Piątek · J. Węglarz
Poznan Supercomputing and Networking Center, Noskowskiego 10 Street, 61-704 Poznan, Poland
e-mail: ariel@man.poznan.pl

K. Kurowski
e-mail: krzysztof.kurowski@man.poznan.pl

W. Piątek
e-mail: piatek@man.poznan.pl

J. Węglarz
Institute of Computing Science, Poznan University of Technology, Piotrowo 2 Street, 60-965 Poznan, Poland
e-mail: jan.weglarz@cs.put.poznan.pl

Fig. 1 A hierarchy of schedulers and typical scheduling phases in grids


End users
Grid scheduler layer
End users jobs in a Grid queue
Job size
Determine jobs requirements and select jobs from a queue (e.g. using FCFS)
Grid broker
dynamic scheduling
Enforce the schedule and monitor jobs
Determine available computing resources
Local scheduler layer
Local scheduler
Local scheduler
Local scheduler
Resource provider(s)
End users jobs in a local queue
Operating system layer
Utilization
Time

they are interested more in a better quality of services provided in grids, where interactive modes and co-allocations are possible using advance reservation features. If we assume that a hierarchical structure of schedulers has to combine strategies with and without advance reservations, many new challenging scheduling problems emerge. We try to address them in this paper.

The paper is organized as follows. Section 2 describes a hierarchical model and methods of scheduling batch tasks without advance reservation. In Sect. 3, a model related to scheduling tasks with advance reservation is presented including state of the art in existing systems, discussion on dynamic binding of reservations, and resource availability representation along with time slot allocation algorithm. Section 4 contains a common model and proposed scheduling policies for workloads consisting of both types of task: those that (i) require and (ii) do not require advance reservation. In Sect. 5, experimental results are presented and discussed. Final conclusions are given in Sect. 6.

2 Hierarchical structure of schedulers

Since most of production grids have been built on the top of the existing local resource management systems (LRMS), a hierarchy of schedulers has been naturally established. At the highest level there is at least one grid-level scheduler that is responsible for management of the workload but it lacks detailed knowledge about the local computing resources where tasks will eventually be run. To understand the complexity of scheduling in a hierarchical structure, one should note that the scheduling process involves several phases at both levels. In particular, they include: determining available computing resources, determining task requirements, invoking a scheduling algorithm which addresses the questions when, which and how many resources are allocated for end-users tasks, and enforcing the schedule and monitoring task execution (see Fig. 1).

2.1 Scheduling without advance reservation

Based on limited knowledge of tasks characteristics, there is a number of queue and space-sharing strategies that may be plugged into schedulers at both grid and local levels. The most commonly used approaches are:

- *First-Come-First-Served (FCFS)* strategy is the simplest approach taking tasks from a waiting queue in the order they arrive
- *Largest-Size-First (LSF)* strategy first takes tasks that require the highest amount of computing resources or, in other words, it sorts the queue according to task-size parameters
- *Short-Job-First (SJF)* executes the shortest task first
- *Preemption policy (PPP)* assumes that all tasks have priorities assigned (e.g. end-users priorities or task-size) that indicate the order in a waiting queue
- *PPP with the dispose procedure (PPPD)* in this policy task priorities in a waiting queue are compared with prior-

ities of running tasks; running tasks with lower priorities are preempted by waiting task with higher priorities

- *Fair-share scheduling (FSS)* resource usage is equally distributed among system users or groups, as opposed to equal distribution among end-user tasks

In practice, the FCFS strategy is still applied frequently in both grid-level and local schedulers. The relatively simple FCFS strategy performs well as long as the computing resources are not heavily used, which is often the case in grid environments. However, if the utilization of computing resources is relatively high other strategies may prove to be at an advantage, especially for end-users and relevant evaluation criteria, e.g. waiting time. The aforementioned scheduling strategies have been mostly applied, tested and compared for schedulers located on local computing clusters or supercomputers, focused solely on the allocation of processors to tasks. To the best of our knowledge, only a few researchers have addressed scheduling problems in a hierarchical scheduling structures that reflect the configuration of resource management systems in typical grid environments (Tchernykh et al. 2006; Kurowski et al. 2008). It is worthwhile to emphasize that the performance evaluation of any scheduling strategy can be considered from different perspectives (criteria) of users (Kurowski et al. 2006a). They use various, often contradictory, evaluation criteria that must be somehow aggregated by a grid-level scheduler to satisfy all the requirements. For instance, resource providers often want to achieve maximum throughput and utilization of computing resources, while end-users expect good performance of their tasks and a short response time. Different stakeholders are not the only reason for taking into account multiple criteria during the resource sharing in grids. For instance, one group of end-users may want their applications to complete as soon as possible, whereas another one is more wiling to wait longer. Typically, the easiest way for users to specify their preferences regarding a level of importance for submitted tasks is a set additional parameters, in particular priorities attached to tasks that can be later used by a scheduler, e.g. using PPP or PPPD strategies. One of the key assumptions in this paper is that many input parameters for scheduling in real grid testbeds are unfortunately missing (Parallel Workload Archive 1999, 2010). In fact, end-users were using relatively simple expressions of task characteristics while interacting with grids in the past according to our analysis of many real grid workloads collected over the last decade. They usually limited a set of task descriptions to a minimum specifying only a basic set of attributes, without for instance trying to estimate critical task characteristics from the scheduling perspective, e.g. task priority, task runtime or advance reservation time slots. Naturally, the more accurate the information that is provided to schedulers during the task submission phase, the more accurate the scheduling decisions can be made.

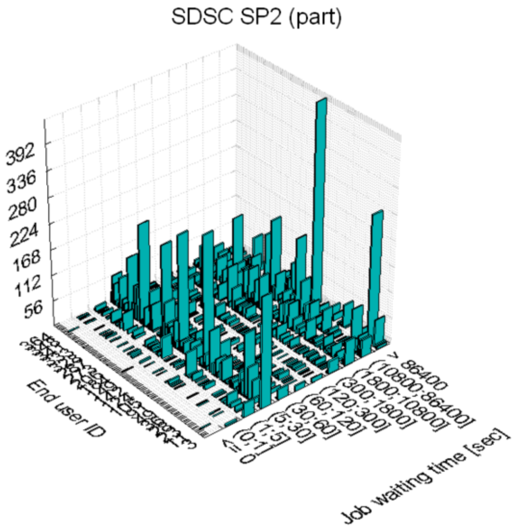


Fig. 2 Different task waiting time periods accepted by end-users in the SDSC SP2 real workload

A simple histogram where tasks were clustered according to waiting time periods are presented in Fig. 2. This is an example real workload of SDSC SP2 system used for our further experiments. It shows that different end-users were using this particular computing cluster in a more interactive way by accepting only relatively small task waiting time periods, whereas there were also end-users tolerating longer task waiting time periods (one should easily note many peaks in the histogram). Unfortunately, there is no information available in the SDSC SP2 workload regarding priorities of tasks and users. We were not able to implement and test more sophisticated PPP, PPPD or FSS strategies for batch tasks without key parameters missing in real workloads. Thus, our experiments and analysis presented in the next sections were focused on FCFS and LSF on-line scheduling strategies for batch tasks. Additionally, we assumed that based on task run times collected in real workloads we were able to generate fixed and flexible advance reservation intervals for tasks with advance reservation requests. Finally, comprehensive studies of multi-criteria approaches to two-level hierarchy scheduling in grids for batch tasks using synthetic workloads were discussed in Kurowski et al. (2008).

3 Scheduling with advance reservation

Nowadays, delivering requested quality of service becomes a more common requirement for new computing infrastructures such as grids and clouds. The quality of service, especially related to an application start time, can be ensured by applying the *advance reservation* functionality which allows users to reserve a certain number of computing resources over time (Kurowski et al. 2006b). We also use the term advance reservation (*AR*) when referring to the resources allocated over the specific time period while computational tasks that require advance reservation to run are called *AR tasks* in this paper. AR is particularly important for the execution of specific classes of applications. Good examples are parallel tasks that require allocation of multiple distributed resources at the same time (to synchronize communication among them), visualization or interactive applications. All of them may need advance reservation to be efficiently and reliably executed. Moreover, advance reservation becomes particularly important if a user has to pay for the utilization of resources. In this case, the user often requires QoS guarantees for a given price, and advance reservation mechanisms can be used to allocate appropriate resources to meet users' requirements.

This section gives a short summary of functionality supporting advance reservation in modern local resource management systems. It also discusses important issues and properties related to AR management, namely dynamic binding of resources and length of time slots used to store ARs. Finally, we propose a data structure and algorithm for searching ARs.

3.1 Advance reservation in existing systems

General concepts and details concerning AR functionality in local resource management systems (LRMSs), or local schedulers, are presented below.

There are several local schedulers that support advance reservation and are widely used to manage both academic and commercial computing clusters, e.g. Platform's Load Sharing Facility (Platform LSF) (Platform LSF 2010), PBS Pro/Torque (PBS 2010), Maui (Maui 2010), and Sun Grid Engine (SGE) (SGE 2010). Maui differs from other systems, as it is a scheduler that extends capabilities of generic LRMSs, for instance, it can be integrated with systems such as Platform LSF, PBS, and SGE. Major functionality related to advance reservation among popular LRMSs are compared in Table 1.

An important issue for systems or users which create ARs is a length of minimal AR time slots. Reservation start and end times are rounded to the start and end times of the minimal AR time slots. For instance, in Platform LSF a length of a minimal AR time slot is 10 minutes, i.e. it is not possible to reserve resources from 12th till 18th minute (instead, a reservation from 10th till 20th minute will be set). The major resource that can be reserved in all systems is a processor (also called a slot). Other resources can be given as constraints (Platform LSF) or can be also reserved if a system is properly reconfigured (SGE). Some systems differ from each other in terms of fundamental approach to AR management. For instance, Platform LSF gives a higher priority to AR tasks by suspending or keeping in a queue non-AR tasks if they might delay any of AR tasks. Different approach is adopted in SGE and relies on wall-clock task runtime limits (queue default if not specified) to determine whether non-AR task can be finished before AR starts. Non-AR tasks without hard wall-clock limit set are never started on a slot which has already a reservation (even in a far future). On the other hand, AR can be created only on a slot that either has no non-AR task running or the non-AR task has runtime limit set. Hence, the efficiency of the whole system depends on accuracy of wall-clock runtime limits. The ability to create advance reservations can lead to potentially unfair and low actual resource usage. This is caused by the fact that a user can reserve all resources in a given system or reserve them in an inefficient way. These drawbacks can be reduced by appropriate policies and limits. Additionally, costs of resource usage limit a size of reservations naturally. LRMSs usually do not support dynamic binding, which means that they reserve specific nodes rather than postpone this decision to the start time of a task. Consequences of this approach are also discussed in Sect. 3.2. Table 1 shows a significant diversity with respect to some LRMS capabilities related to advance reservation, with particular focus on interaction between AR and non-AR tasks. More detailed comparison of advance reservation in LRMSs can be found in Oleksiak (2009). Most of of the systems provide rather limited functionality. For instance, none of the considered LRMSs addresses more complex queries about reservations such as an interval search function. Instead, they just return answer "yes" or "no" concerning specific advance reservation.

Nevertheless, some of these differences as well as limited capabilities can be hidden by resource providers responsible for managing local systems and delivering remote access to LRMSs. An example of such resource provider is SMOA Computing (SMOA 2010) that exposes the AR capability of underlying LRMSs (Platform LSF, SGE). In addition to basic capabilities, it enables creating the so-called *best effort* reservations. The best effort reservations allow LRMS to reserve either available number of slots within a range given by a user (per-slot best-effort) or a specific number of slots for available time also from a range given by a user (per-time best-effort).

Many of the capabilities summarized in Table 1 may significantly impact efficiency of AR management and the whole system. Thus, in the next sections we discuss them and propose solutions that we applied in our work.

3.2 Dynamic binding of reserved resources

Most of the data structures for storing ARs described in the literature is based on the assumption that a number of available resources in a certain time period is sufficient to determine whether a new advance reservation can be created

Table 1 Comparison of advance reservation capabilities of local schedulers

Property	Platform LSF	PBS Pro	Maui	SGE
Granularity of AR time slot	10 minutes	Seconds	Seconds	Seconds
Behavior of active tasks on AR end (before completion of all tasks)	Tasks killed (if AR is open tasks are suspended)	Tasks killed	Tasks killed	Tasks killed (by default 60 s before AR end)
Impact of AR on non-AR running tasks	Non-AR task suspended if it blocks AR task resumed when AR ends	No impact (runtime limit used see below)	No impact (runtime limit used see below)	No impact (runtime limit used see below)
Impact of AR on non-AR waiting	Non-AR task remains pending if ARs use all slots	Non-AR task started if its runtime limit less than AR start time	Non-AR task started if its runtime limit less than AR start time	Non-AR task started if its runtime limit less than AR start time or (if runtime limit not set) no ARs on nodes
Impact of non-AR tasks on creation of AR	No impact (see above)	Runtime limits taken into account	Runtime limits taken into account if proper priorities and backfilling ARs not delayed	AR only on slot without non-AR tasks running unless they have runtime limits
Input to query about AR	Start and end time, number of CPUs, resource constraints	Start and end time, resource list	Start and end time, amount of resources, resource constraints	Start and end time, number of slots, resource constraints
Dynamic binding	AR created on concrete nodes	Required resource are limits of AR queue	AR created on arbitrary or concrete nodes specified by AR owner	AR slots on concrete nodes

within this period. However, when ARs are strictly assigned to specific resources before start of a task, this assumption may not be met. The specific condition that must occur to meet this assumption is precisely defined by the following proposition.

Proposition 1 *If specific resources for all ARs are allocated not earlier than at their start times t^s (using the so-called dynamic binding) then information about the amounts of free resources over time is sufficient to determine whether the AR can be created.*

Let us assume that according to the data structure that contains information about amounts of available resources over time, an advance reservation AR_x of size *size* can be allocated from t^s to t^e. Hence, for any time $t \in \langle t^s, t^e \rangle$, a sum of *size* and resources allocated by all other ARs at time t does not exceed the initial amount of resource available, m_0. Resources allocated by all other ARs at time t can be calculated as a result of subtraction of a sum of resources allocated by ARs started before t and a sum of resources released by ARs ended before t. This observation, illustrated in Fig.3, can be defined in the form of the following equation:

$$\forall_{t \in (t^s, t^e)} \left(\sum_{i:t_i^s < t} size_i - \sum_{j:t_j^e < t} size_j + size \le m_0 \right) \tag{1}$$

According to the assumption of the proposition, AR cannot be blocked by later ARs (with greater start times than t^s) because AR_x can be allocated first (due to assumed dynamic binding property). For instance, AR_3 in Fig. 3 cannot prevent allocation of AR_x because resources for AR_3 will be allocated later. Hence, according to (1), AR_x can be allocated any time $t \in \langle t^s, t^e \rangle$. For start times less than the start times of other ARs ($t < t_3^s$ in the case of Fig. 3) any free processors can be chosen. After allocation of AR_x AR_3 can be allocated too. If it could not be allocated, that would be contradictory to (1). This fact is a basis of proof presented in Oleksiak (2009).

In consequence, if a resource provider does not dynamically bind resources to ARs at their start times (i.e. does not support the so-called *dynamic binding* functionality) then the description of resource availability may not be sufficient to determine if AR can be created. Unfortunately, LRMSs do not rather support dynamic binding (see Sect. 3.1). Therefore, resource providers built on top of existing LRMSs can

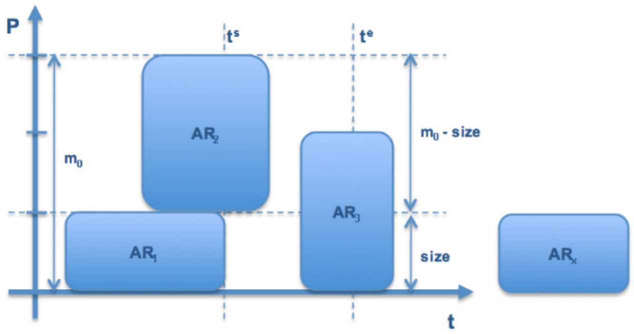


Fig. 3 Inserting advance reservation AR_x: illustration of Proposition 1

implement this feature and then provide it to the upper grid layer.

Most of the approaches (e.g. Burchard 2005; Sulistio et al. 2007; Brodnik and Nilsson 2003) limit information about slots to a number of available computing resources (usually processors or computing nodes) skipping the knowledge about concrete identifiers of these resources. In other words, they assume that for a certain task which requires m processors, if in subsequent slots at least m processors are available, then the task can be executed. However, this assumption may be false if there are not *m the same* processors available in subsequent slots, since we consider non-preemptive tasks. In this case, the sets of processors must be compared (not only their total number). We proposed an extended data structure with specific resources information (e.g. processors) in Kurowski et al. (2010). In this paper we focus on scheduling with dynamic binding as it facilitates scheduling process for a hierarchical scheduler and positively impacts efficiency of scheduling both types of task: AR and non-AR tasks.

3.3 Variable-length time slots

The approaches referenced in the previous section use slots of the same size. Although this assumption helps to achieve a good performance as reported in Sulistio et al. (2008), it might have important drawbacks. Namely, if a slot length is too small, it impacts the performance which, as might be expected, depends on a number of slots. Large numbers of slots must be managed, in particular, if periods are long and many providers are considered. This issue becomes particularly cumbersome if information about resource availability is exchanged between distributed systems, and this is the case in our model. On the other hand, too long slots may cause significant resource under-utilization, as some long slots may remain idle. It is difficult to set a slot length correctly in order to obtain a trade-off of utilization and performance, especially that these measures depends on task lengths. To avoid this problem the variable-length slots can be used. Their length depends on specific advance reservations that were allocated. Our approach to construct and manage such slots is presented in the next section. The influence of the slot length on resource utilization for the fixed- and variable-length slot representation was studied experimentally (see Figs. 4 and 5 for details). First, we studied the impact of particular methods on quality of results, namely, resource utilization, makespan, and mean flow time. We compare the change-based, or variable, slots approach with two versions of the fixed-length slots approach: with short (10 minutes) and long (1 hour) slot length. Resource utilization is compared in Fig. 4. It is easy to see that the longer length of a fixed slot, the bigger the number of “gaps” that lead to lower resource utilization.

The results presented above are confirmed in Fig. 5 where our expectations concerning resource utilization are confirmed by precise values. They are complemented by values of makespan and mean flow time.

Additionally, most of the existing methods do not provide interval search operation, in which possible execution times are searched within a given interval instead of checking a specific time. Finally, none of these approaches considered searching for availability of specific resources (instead of their amount only) although this issue may be important as was shown in Proposition 1. Therefore, we decided to introduce new data structures and algorithms to store and manage resource availability time slots.

3.4 Data structure for storing ARs

Several approaches to manage ARs were proposed to date (Burchard 2005; Sulistio et al. 2008; Brodnik and Nilsson 2003). However, these methods have disadvantages that prevented us from using them directly for our purpose. Due to the observations made above, as well as the results of experiments, we decided to use a structure with variable slot lengths. Unfortunately, approaches for variable slots that already exist, such as the linked list (Xiong et al. 2005), are characterized by rather low performance. Therefore, we propose a new representation that meets requirements of our model. We called this structure the change-based advance reservation time slots (*CARTS*). We assume that each slot represents a period in which availability of resources does not change, i.e. either no reservation starts or finishes or the same amount of resources is released by ending reservations and allocated by starting ones. In this way, each slot provides information concerning the available amount which is constant for this slot. Slots are sorted by increasing start times. Since end time of each slot is equal to the start time of the next slot, it is sufficient to store a single value of a start time for each slot. It is also easy to use a binary search to locate the earliest and latest slots with respect to users' requests. *CARTS* provides a description of resource availability required by a grid scheduler to create feasible and

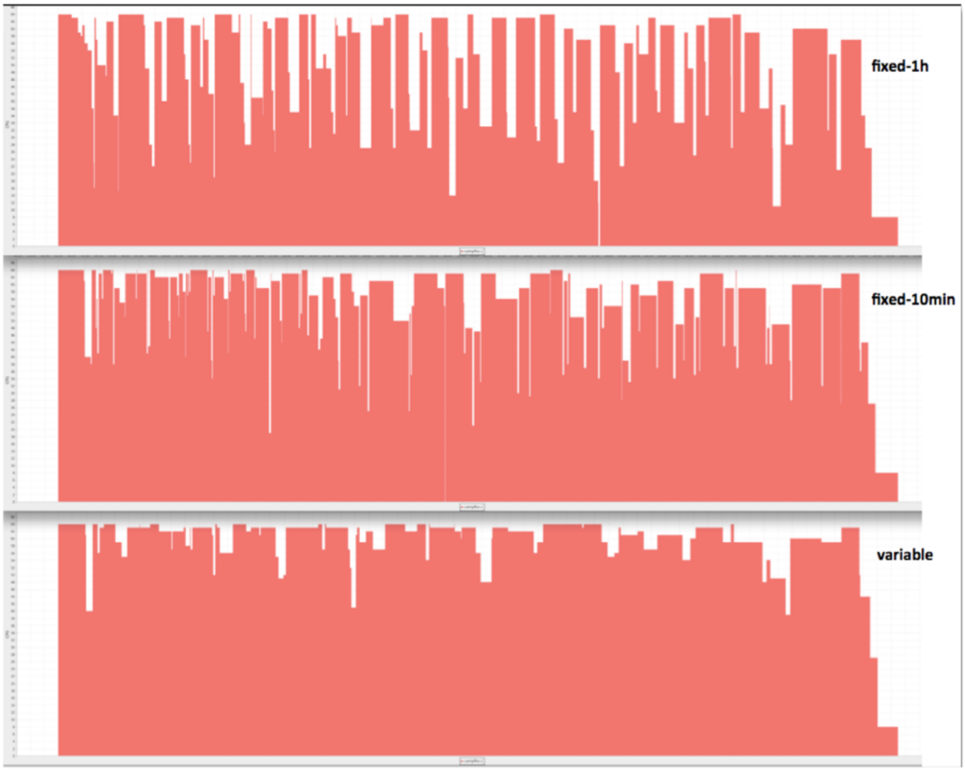


Fig. 4 Resource utilization over time for the fixed- (*two upper charts*) and variable-length (*chart at the bottom*) slots

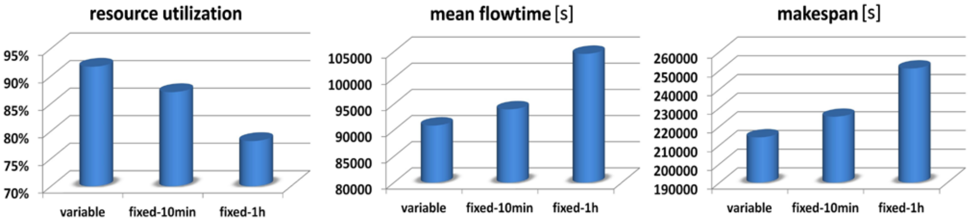


Fig. 5 Resource utilization, makespan, and mean flow time for variable- and fixed-length slots approaches

efficient schedules. *CARTS* structure is a list of subsequent resource availability time slots RT, or simply slots.

$RT = \{RT_1, \ldots, RT_v\}$, $RT_l = (t_l^0, m_l), l = 1, \ldots, v$, where

- v is a number of time slots
- t_l^0 is a start time of slot l
- m_l is a number of resources (processors)

According to Proposition 1 for systems with dynamic binding, the *CARTS* data structure assumes that a number of available resources in a certain time period is sufficient to decide whether a new advance reservation can be created within this period.

Below we propose an algorithm which generates the *CARTS* structure taking as an input a list of advance reservations. More precisely, it takes two arrays: AR start times and end times sorted in nondecreasing order. Basically, the algorithm iterates through the two arrays and adds slots to the *CARTS* structure for all changes of resource availabilities. In other words, it checks whether a difference between a sum of resources allocated at a specific time by starting ARs and a sum of resources released at this time by ending ARs is equal to zero. If not, resource availability changed and a new slot must be added to describe it.

Algorithm 1 is of complexity $O(n)$, where $n = |AR|$ (i.e. a number of reservations), assuming nondecreasing order of reservation start and end times. If they were not kept sorted, the complexity would rise to $O(n \log n)$. In this algorithm we assumed that the *CARTS* structure is generated based on all reservations. However, to obtain resource availability in a requested time period, the algorithm would have to take as input reservations that have not finished before this period. To this end a binary search can be used to find the first reservations (see Algorithm 2 in Sect. 3.5 for details). The search would cost $O(\log n)$ operations so it would not increase the complexity of the algorithm. Additionally, algorithm loops would need to include stop conditions related to the end of the requested time period.

3.5 Resource availability time slot searching algorithm

In this section we present the AR time slot search algorithm (*ARTS*^{2}) using *CARTS* structure. This algorithm searches for the earliest possible time slot for a requested AR in a given interval. A detailed algorithm of searching for available slots in the *CARTS* structure is presented below. In its first step the earliest slot that meets user's time requirements is found. This step can be done in $O(\log v)$ time, where v is a number of slots (dependent on a number of ARs—in more detail $v \leq 2|AR|$). Then it is easy to check if an AR request fits to several subsequent slots. If for any slot i a number of available processors m_i is lower than a number of processors requested by AR, this AR cannot be created. A detailed algorithm is presented as Algorithm 2.

As was shown in Proposition 1 in Sect. 3.2, if a resource provider supports the dynamic binding capability, then a description of resource availability is sufficient to determine if AR can be created. However, if dynamic binding is not supported one cannot assume that information about the amount of free processors will allow to decide about the allocation of AR. In such a case information about availability of specific processors must be given in order to perform scheduling. An extended *CARTS* structure with specific resources information (e.g. processors): *CARTS* − *SR* was proposed in Kurowski et al. (2010).

Of course, good quality of results (as studied in Sect. 3.3) is not the only requirement for the data structures used to represent resource availability (or advance reservations). Another one is short processing time, in particular, interval searching for free time slots.

The results of performance tests showed that the method based on variable slots provided search times comparable to the fixed-slot approach with moderate slot length (up to (Oleksiak 2009). The fixed-slot approach outperformed the variable-slot one for longer slots, however, led to a significantly lower resource utilization. On the other hand, short fixed-length slots (around 10 min) caused long search times, longer than in the case of variable slots. Searching variable-length slots worked particularly fast for workloads consisting of many big reservations.

4 Common model for scheduling both AR and non-AR tasks

Scheduling of both typical batch tasks and advance reservations in a single resource management system is a challenge, especially if a hierarchical scheduling structure is considered. As it was presented in Sect. 3.1, ARs heavily impact utilization and execution of remaining tasks in the system. Similarly, non-AR tasks may also significantly influence setting ARs. Due to these issues, administrators of LRMSs often prefer to avoid advance reservations. Therefore, we attempt to provide a model and method which could efficiently

Algorithm 1: Generation of CARTS

```
Input:
// Array of sorted AR start times
$(t_1^s, t_2^s, \ldots, t_{|AR|}^s)$;
// Array of sorted AR end times
$(t_1^e, t_2^e, \ldots, t_{|AR|}^e)$;
// Sizes of ARs sorted by start
   times
$(size_{t_1^s}, \ldots, size_{t_{|AR|}^s})$;
// Sizes of ARs sorted by end times
$(size_{t_1^e}, \ldots, size_{t_{|AR|}^e})$;
Output:
// CARTS
$CARTS = ((t_1, m_1), (t_2, m_2), \ldots, (t_v, m_v))$;
$t_0 \leftarrow 0$;
// Number of processors available at
   the beginning
$m_0 \leftarrow m$;
$v \leftarrow 1$;
$i \leftarrow 1$;
$j \leftarrow 1$;
// not all reservations checked
while $i \leq |AR|$ OR $j \leq |AR|$ do
    // start times before next end
       time
    while $t_i^s \leq t_j^e$ OR $i \leq |AR|$ do
        if $t_i^s > t_v$ then
            // subsequent slots differ
            if $m_v \neq m_{v-1}$ then
                // so next slot added
                $v \leftarrow v + 1$;
                $m_v \leftarrow m_{v-1}$;
            end
            $t_v \leftarrow t_i^s$;
        end
        // resources allocated by AR i
        $m_v \leftarrow m_v - size_{t_i^s}$;
        $i \leftarrow i + 1$;
    end
    // end times before next start
       time
    while $t_j^e \leq t_i^s$ OR $j \leq |AR|$ do
        if $t_j^e > t_v$ then
            // subsequent slots differ
            if $m_v \neq m_{v-1}$ then
                // so next slot added
                $v \leftarrow v + 1$;
                $m_v \leftarrow m_{v-1}$;
            end
            $t_v \leftarrow t_j^e$;
        end
        // resources released by AR
        $m_v \leftarrow m_v + size_{t_j^e}$;
        $j \leftarrow j + 1$;
    end
end
return $((t_1, m_1), (t_2, m_2), \ldots, (t_v, m_v))$;
```

Algorithm 2: Time slot search algorithm ($ARTS^2$)

```
Input:
// interval given by user (ready
   time and deadline)
$\langle r, d\rangle$;
// AR duration given by user
   (processing time)
$p$;
// number of processors requested by
   user
$size$;
// CARTS
$RT_i = (t_i, m_i), i = 1, \ldots, v$;
Output:
// earliest start time of requested
   AR
$t_{start}$;
// $i_{start} = \arg\min_i \{t_i : t_i \geq r\}$
$i_{start} \leftarrow$ binary_search($r$, 0, $v$);
$i \leftarrow i_{start}$;
$length \leftarrow 0$;
$count \leftarrow 0$;
repeat
    // not enough processors in a slot
    if $size > m_i$ then
        // set start to subsequent slot
        $i_{start} \leftarrow i + 1$;
        length $\leftarrow 0$;
        count $\leftarrow 0$;
    else // enough processors in a slot
        $length \leftarrow length + (t_{i+1} - t_i)$;
        $count \leftarrow count + 1$;
    end
    $i \leftarrow i + 1$;
until ($length \geq p$) OR ($t_{i_{start}} + p > d$) OR $i \geq v$
// AR can be created or deadline is
exceeded or there are no more slots;
$t_{start} \leftarrow t_{i_{start}}$;
```

manage both types of task. We assumed (based on solutions applied in most of LRMSs) that for all tasks the processing time limit is set. This limit can be set by end-users during task submission or, if not, by LRMS according to configuration of particular queues. Length of advance reservations is given by users.

In this section we specify information about computational tasks and resources available for grid and local schedulers. As presented in Sect. 2 we consider a hierarchical model in which we assume that computational tasks are submitted to a grid scheduler that dispatches them to local schedulers. Then, local schedulers generate their schedules for resources they manage. The schedulers must deal with both advance reservations and regular (i.e. best effort) tasks. The following paragraphs present definitions of model parameters related to tasks, resources, and evaluation criteria.

4.1 Computational tasks

We consider two separate sets of tasks that arrive to a grid scheduler:

- J—*Set of computational tasks*. It contains regular tasks that do not require advance reservation
- J^R—*Set of advance reservation requests*. It contains tasks that require advance reservation

Thus, particular tasks are denoted as follows:

- j_i—*Computational task i* (without reservation request), $i = 1, \ldots, |J|$. In short *non-AR task*
- j_k^R—*Advance reservation request k* (computational task with reservation request), $k = 1, \ldots, |J^R|$. In short *AR task*

Each computational task is characterized by the following parameters:

- s_i—*Size* is a number of processors required by task j_i. Since we consider only *rigid tasks* in our model, we assume that s_i is constant for each task, i.e. it can change neither before execution of a task (during scheduling) nor during execution
- p_i—*Processing time* is the time needed to execute task j_i. In our model we assume that processing times of non-AR tasks are not known by schedulers. This assumption is based on observation of real systems. Instead, task execution time limits can be used. See parameter c_i for more details
- c_i—*Processing time limit* is the maximum execution time of task j_i. If processing time of non-AR task exceeds c_i, it is stopped. Processing time limits (called also requested time) are commonly used in LRMSs in backfilling and to allocate tasks before advance reservations. Processing time limits are given by users or, if not, imposed by configuration of queues

In addition to parameter of regular tasks, the AR tasks are defined using the following parameters:

- r_k—*Ready time* is the time at which task j_k^R is ready for processing. Ready time of a task results from the earliest start time defined in the time requirements by a user
- d_k—*Deadline* is the latest allowed AR task completion time. Deadline d_k is a hard real-time limit by which j_k^R must be completed. Hence, a pair $\langle r_k, d_k\rangle$ specifies a time period within which task j_k^R must be scheduled. Important note: in order to ensure fair comparison of mean utilization and flow time d_k is treated as if it were a due date in most of experiments. Thus, instead of number of rejected ARs the tardiness of AR tasks is calculated

Both types of task are rigid and non-preemptable.

4.2 Resources

The grid scheduler submit its tasks to multiple resource providers that run local schedulers to manage their resources. Resource providers and their resources are defined by the following parameters:

- R—*Set of resource providers*
- R_l—*Resource provider* l, $l = 1, \dots, |R|$
- N_l—*Set of processors* of resource provider R_l
- n_{lm}—*Processor* m at resource provider l, $m = 1, \dots, |N_l|$
- q_l—*Non-AR tasks queue* at resource provider R_l
- q_l^R—*AR tasks queue* at resource provider R_l
- q—*Grid scheduler queue*

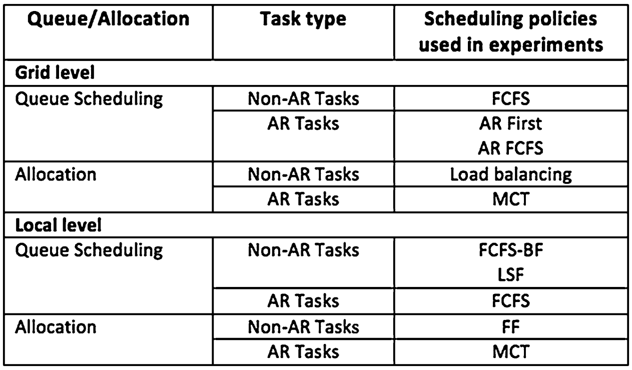
In this paper we assume that resources are homogeneous. We also assume that each LRMS has one queue for non-AR and one for AR tasks. There is one global queue at the grid scheduler level where all tasks, non-AR and AR, arrive.

4.3 Criteria

Scheduling a diverse set of grid tasks with and without advance reservations requires consideration of multiple points of view. First of all, resource administrators (and owners) are interested in maximization of resource utilization. Users submitting their (non-AR) tasks to queues aim at short waiting times. Finally, users requesting advance reservations require that their reservations are not rejected nor delayed. To address these points of views, we studied an impact of scheduling policies and parameters on three major criteria: resource utilization, mean flow time, and mean tardiness. Before we define these criteria, let us introduce needed parameters:

- S_i—*Start time* of task j_i defines a start time chosen by a scheduler
- C_i—*End time* of task j_i defines a real end time (task completion time)
- *Task flow time*: $F_i = C_i - r_i$, where r_i is a ready time of task j_i
- *Task lateness*: $L_k = C_k - d_k$, where d_k is a due date of task j_k^R, this metric is calculated for AR tasks (for non-AR tasks the lateness is equal to zero)
- *Task tardiness*: $D_k = \max(0, C_k - d_k)$, this metric, as in the case of lateness, is calculated for AR tasks (for non-AR tasks the tardiness is equal to zero)
- *Resource provider utilization*: $U_l = \frac{\sum_{i=1,\dots,|J_l|} s_i * p_i + \sum_{k=1,\dots,|J_l^R|} s_k^R * p_k^R)}{C_{\mathrm{MAX}} * |N_l|}$, where J_l and J_l^R are sets of non-AR and AR tasks executed at resource provider R_l, respectively.

Having the parameters presented above, the three evaluation criteria considered in this paper can be defined as follows:



Queue/Allocation	Task type	Scheduling policies used in experiments
Grid level		
Queue Scheduling	Non-AR Tasks	FCFS
	AR Tasks	AR First AR FCFS
Allocation	Non-AR Tasks	Load balancing
	AR Tasks	MCT
Local level		
Queue Scheduling	Non-AR Tasks	FCFS-BF LSF
	AR Tasks	FCFS
Allocation	Non-AR Tasks	FF
	AR Tasks	MCT

Fig. 6 Scheduling policies in a hierarchical grid system

Mean flow time:

$$\textit{mean } F = \frac{1}{|J| + |J^R|} \left(\sum_{i=1,\dots,|J|} F_i + \sum_{k=1,\dots,|J^R|} F_k \right) \tag{2}$$

Mean tardiness:

$$\textit{mean } D = \frac{1}{|J^R|} \sum_{k=1,\dots,|J^R|} D_k \tag{3}$$

Mean resource utilization:

$$\textit{mean } U = \frac{1}{|R|} \sum_{l=1,\dots,|R|} U_l \tag{4}$$

4.4 Strategies

In hierarchical grid systems grid and local schedulers use separate scheduling strategies. As stated in previous sections, we assumed that tasks arrive on-line, as in most grid systems encountered in practice. Thus, for both grid and local schedulers, we define in the following paragraphs queuing and allocation strategies. The former is responsible for setting order of tasks in queues while the latter determines about allocation of tasks to specific resources. These policies usually differ for tasks that require advance reservation as well tasks that do not. This distinction is also reflected in the following paragraphs. Scheduling policies used in our experiments are summarized in Fig. 6. Independently from scheduling policies a form of AR requests impacts the scheduling process. Therefore, before describing the scheduling policies, we present in the next paragraph two types of AR request that we considered in this paper.

4.4.1 AR request

By an *AR request* we mean requirements concerning AR defined by a user during the submission phase. As presented in Sect. 3 it consists of start and end time of reservation k (t_k^s and t_k^e, respectively) and a number of CPUs to reserve

($size_k$). Additionally, we consider two types of AR request: *fixed* and *flexible*.

A fixed AR request denotes fixed values of requested AR start and end time that are equal in this case to ready time and deadline of advance reservation, i.e. $t_k^s = r_k$ and $t_k^e = d_k$.

A flexible AR request denotes flexible values of requested AR start and end times, greater or equal to ready time and less or equal to deadline, respectively: $t_k^s \geq r_k$ and $t_k^e \leq d_k$.

Flexible AR requests give more flexibility to the scheduler allowing it to adjust start time of reservations to other tasks. Flexible AR requests also decrease a number of rejections. It is important to note that in most experiments we treated deadline d_k as a due date in order to calculate values of tardiness criterion and to have a fair comparison of all values for the criteria. Otherwise, in case of rejections it would be hard to compare other criteria: flow time and resource utilization (as a number of completed tasks may change).

4.4.2 Grid level policies

Queue scheduling

We assume there is a single grid scheduler queue q that contains two types of task: with reservations (*AR*) and without reservations (*non-AR*). Generally, the grid-level scheduler can apply different scheduling policy to each of them.

Non-AR tasks For non-AR tasks we applied the First Come First Served (FCFS) method. This simple (and usually inefficient) method was used because the grid-level scheduler cannot use backfilling or more advanced scheduling since tasks are sent to queues of local schedulers and scheduled according to their own policies. Thus, tasks are taken from a queue in order of their arrival and allocated to resources (according to a method presented in the next paragraph). The decision about new task to allocate is taken every time any task finishes or new task arrives.

AR tasks For AR tasks two options were applied:

- *AR first*. AR requests are forwarded to local schedulers whenever they arrive to the grid scheduler. In other words AR tasks have higher priority than non-AR tasks
- *AR FCFS*. AR tasks are scheduled in the same way as non-AR tasks. This means that AR tasks have the same priority as non-AR tasks

Allocation to resource providers

Non-AR tasks In case of batch tasks that are submitted to queues of local schedulers the grid scheduler cannot directly impact the start time of a task. It usually selects a local scheduler (resource provider R_l) so that the total load is balanced. We applied a load balancing policy based on a number of tasks in queues of local schedulers. For every task j_i the resource provider R_l with minimum queue length is selected.

AR tasks To select a resource provider for AR tasks we used two methods.

- *MCT*. The Minimum Completion Time (MCT) policy allocates a tasks to a resource that guarantees the earliest completion time. In our case, as resources are homogeneous, MCT means finding the earliest possible start time. Thus grid scheduler selects for a given AR task the resource provider R_l with a minimum earliest start time
- *AR resources priority*. In the second method we introduced prioritized resource providers for AR tasks. In more detail, we defined a list of resource providers, sorted by decreasing priority for AR tasks. The policy starts from the first resource provider from the list and sends AR request until it gets positive answer. The main difference to the MCT policy is that resource providers are prioritized and an AR task is submitted to the resource provider with a higher priority even if a lower priority provider offers earlier start time. The sufficient condition is that a resource provider must meet requirements specified in the AR request. This approach was introduced to group ARs together and reduce their negative impact on non-AR tasks. It is worth to note that grouping is done in a dynamic and relatively simple way, on the contrary to a common approach in practice with a static subset of resources dedicated for ARs.

4.4.3 Local level policies

Queue scheduling

Non-AR tasks To select a computing node for non-AR tasks we used two methods:

- *FCFS-BF*. First we applied the well-known and commonly used First Come First Served (FCFS) method with backfilling (FCFS-BF). Tasks are taken from a queue in order of their arrival and allocated to resources (according to a method presented in the next paragraph). If a task cannot be allocated at a given moment due to lack of resources the next task from the queue is checked. The decision about a new task to allocate is taken every time any task finishes or a new task arrives
- *LSF*. Another method that we adopted was Largest Size First (LSF). To this end, tasks were sorted from the largest (the biggest $size_i$) to the smallest

AR tasks At the local scheduler AR tasks are scheduled when AR request arrives and selected schedule is accepted by the grid scheduler. At AR start time t^s AR tasks are allocated to specific nodes (see next paragraph).

Allocation to nodes

Non-AR tasks To allocate non-AR tasks to specific nodes the First Fit (FF) algorithm was used. The scheduler uses the processing time limit c_i to check if a task can be allocated. If AR starts earlier than estimated non-AR completion time (sum of start time and processing time limit) a task does not fit and cannot be allocated. Thus, a task j_x fits if for all non-AR tasks running on nodes that are needed by j_x: $S_i + c_i \leq S_x$. Processing time limits are usually significantly longer than the actual task processing time often leading to idle periods before reservations. Therefore, it is better to avoid allocations of AR after non-AR tasks which justifies our approach to group AR tasks, as described in previous paragraphs.

AR tasks. The scheduling AR tasks is done based on a number of available nodes and using the MCT algorithm at the moment when the AR request arrives. The allocation of AR tasks to specific nodes is done at AR start time t^s (which is called dynamic binding) looking for free nodes starting from a node with the lowest index. To check if nodes are free the processing time limits of running non-AR tasks are used similarly as in the allocation of non-AR tasks.

4.4.4 *On-line and off-line scheduling period*

In general, scheduling strategies can be restricted to two generic scheduling models, where either *all tasks arrive at time zero*, or *tasks can arrive dynamically at any arbitrary time*. In this paper we present advanced analyses of different hierarchical scheduling configurations of on-line *open* and on-line *closed* systems for non-AR and AR tasks. In the first case we consider a system with an endless stream of tasks arriving to the grid queue which are immediately forwarded to underlying queues. In the second approach we assume that a set of tasks is taken from the grid-level queue periodically. In other words, a set of tasks at a certain point in time are assigned by a grid-level scheduler to underlying queues according to a scheduling strategy, for instance Random or LB. At local-level different scheduling strategies can be applied as well, e.g. FCFS or LSF. A period of time when the grid-level scheduler collects an updated information from all resource providers about dynamic characteristics of tasks and resources is also called *off-line scheduling period* or *passive scheduling phase* and denoted as δ. During the off-line scheduling period end-users tasks constantly arrive to the grid-level scheduler and are placed in the grid-level queue. If $\delta = 0$ than the scheduling procedure is invoked by the task submission event, and the strategy is classified as *on-line* or *active scheduling phase*. For instance, in contrast to FCFS, LSF is an example of the off-line strategy which is invoked periodically according to defined frequency δ. LSF sorts the queue according to the task-size parameter (a number of required computing resources). Obviously, the longer system waits to apply the local scheduling strategy the higher number of tasks is collected for further processing. This natural characteristic of any off-line scheduling strategy influences the overall performance of the system, especially all evaluation metrics based on flow and waiting time parameters as there are no scheduling decisions made during the off-line period. On the other hand, a set of collected tasks together with their requirements, in particular tasks requiring advance reservations, allows a scheduler to take into account simultaneously all task and resource characteristics to better optimize schedules.

5 Experimental results

All our experiments were performed using GSSIM—the simulation environment for hierarchical scheduling systems which we present in Kurowski et al. (2007). First, in Sect. 5.1 we study the performance of typical scheduling strategies for the two-level scheduling structure without AR for different configurations using the real workload SDSC SP2 available at Parallel Workload Archive (1999, 2010). The selected workload is one of the commonly used workloads by researchers, e.g. in Aida and Casanova (2009), Feitelson (2008), or Iosup et al. (2008). Then, in Sect. 5.2 we compare various scenarios in simulation with a different number of non-AR and AR tasks to demonstrate their impact on schedules. Characteristic of the workload used in the experiments is also discussed. In Sect. 4.4.4 a study of the impact of the off-line scheduling period on evaluation criteria is presented. This section also discusses the partitioning of resources in LRMSs. Section 5.4 contains an evaluation of the proposed AR Resource Priority strategy compared with common MCT approach. Finally, in Sect. 5.5 we perform an experimental study of hierarchical scheduling strategies in the light of defined three evaluation criteria.

5.1 Performance evaluation of scheduling strategies for non-AR tasks

In order to perform comprehensive simulation studies of different scheduling strategies for non-AR tasks we used a two-level hierarchical scheduling structure introduced in Sect. 2. We experimentally tested and compared FCFS (on-line scheduling strategy) and LSF (off-line scheduling strategy). First, we ran some experiments for scheduling strategies at the local level by invoking the LSF strategy every $\delta = 10$, $\delta = 60$, $\delta = 600$ and $\delta = 3600$ seconds. As presented in Table 2, the FCFS strategy performed quite well, but it was outperformed by LSF-10 and LSF-60 corresponding to the scheduling configuration of LSF with the off-line scheduling period $\delta = 10$ and $\delta = 60$ seconds. All three strategies FCFS, LSF-10 and LSF-60 reached the same level of utilization *mean* $U = 0.58$. Nevertheless, the total waiting

Table 2 Performance of local scheduling strategies: FCFS and LSF applied for the SDSC SP2 workload (tasks 1–1000)

Queues × Res.	Grid-level	Local-level	*mean U*	*mean F*	Total waiting time
1 × 128	FCFS	FCFS	0.587	16228	8056
1 × 128	FCFS	LSF-10	0.587	15314	7142
1 × 128	FCFS	LSF-60	0.587	15956	7783
1 × 128	FCFS	LSF-600	0.58	19051	10879
1 × 128	FCFS	LSF-3600	0.532	141727	133555

time was reduced by 13% in the case of LSF-10 and 2% in the case of LSF-60. As expected, much longer invocation periods (δ) applied in the LSF strategy affected significantly two evaluation criteria: *mean F* and the corresponding total waiting time. The obtained results on those two criteria increased up to 141727 and 133555, respectively, for $\delta = 3600$ (LSF-3600). One should also note that in practical settings much longer invocation periods (e.g. $\delta >$ a few hours) of any off-line scheduling strategy are not acceptable for administrators and end-users. However, for relatively short δ there is a great opportunity to introduce new off-line optimization techniques to improve scheduling strategies as we demonstrated in Kurowski et al. (2008).

Additionally, we decided to introduce more advanced hierarchical scheduling structures, where many local queues were involved at the local-level. In new tests we used a relatively simple Random procedure at grid-level.

First, we had to partition computing resources into a few sets based on additional descriptions and comments typically attached to real workloads at Parallel Workload Archive (1999, 2010). In practice, partitioning techniques are often used by local administrators giving them the possibility of assigning end-users tasks or computational resources to various queues depending on past users behaviors. In this case, the number of computing resources reported for SDSC SP2 was 128, and we aimed at creating two partitions providing the access through two queues to 2 × 64 computing resources as is described in Table 3. Thus, we had to modify the original SDSC SP2 workload by reducing the maximum number of requested computing resources from 128 to 64 for tasks identified as: 86, 91, 95, 144, 171 (end-user 92), and 208, 742, 746, 750, 920, 967 (end-user 147). Then, we created two-level hierarchical structures for four and eight queues adding the same portion of 64 computing resources to our simulations. Partitioning the SDSC SP2 system into two parts (using the same real workload) resulted, as expected, in a lower utilization of computing resources to less than *mean U* 60%. Schedulers obtained by FCFS during our experiments were satisfactory taking into account previous analysis of real systems indicating a typical utilization of 60–80% for much more advanced scheduling techniques than FCFS, e.g. conservative backfilling presented in Jones (1999). Additionally, we also discovered experimentally the best configuration of Random and LSF procedures for the invocation period $\delta = 600$ (see Table 3). Much longer δ parameters for LSF resulted in worse performance on two criteria relevant for end-users, namely *mean F* and total waiting time.

Table 3 Performance of two-level hierarchical scheduling structures for the SDSC SP2 workload

$Q \times R$	Grid-level	Local-level	*mean U*	*mean F*	Waiting time
2 × 64	Random	FCFS	0.592	44559	38512
4 × 64	Random	FCFS	0.391	17481	10259
8 × 64	Random	FCFS	0.203	9865	2579
2 × 64	Random	LSF-600	0.644	64175	58224
4 × 64	Random	LSF-600	0.391	15287	8151
8 × 64	Random	LSF-600	0.209	10675	3287

5.2 Impact of AR tasks on efficiency of schedules

In order to evaluate the impact of workload characteristics on performance of scheduling strategy we used the same SDSC SP2 workload. However, for experimental purposes, we extracted two separate sets of tasks from the original workload. The first set contains tasks with IDs from 0 to 1000, whereas the second one contains tasks with IDs from 68000 to 69000. Their basic characteristics are given in Table 4. The main difference between them is related to the number of requested processors and inter arrival time. In addition, since the original real workload does not provide information about AR tasks, we chose a given percent of tasks and treated them as AR tasks. The selection was made using discrete uniform distribution. Subsets of AR tasks are equal in repetitions of specific experiments. Performing experiments related to the fixed AR intervals required setting interval search time to specific values. We assumed that AR tasks have the start time equal to task submission time + waiting time. The duration of such task is based on its requested runtime. These values are read directly from the workload. On the other hand, flexible AR intervals assume that we extend the interval of a reservation query. Thus, the book ahead time is enlarged by the given parameter, as well as the search limit time. In this way we allow selection of the specific AR start time within a given interval. The intervals used in the experiments are presented in Sect. 5.4.2.

Let us now show example schedules generated by GSSIM including 50% of AR tasks where two separated queues with 64 computing resources were used. Generated schedulers are illustrated in the form of Gantt charts in Fig. 7. One can see the gaps before ARs that lower the total resource utilization and impact non-AR tasks. More detailed analysis of

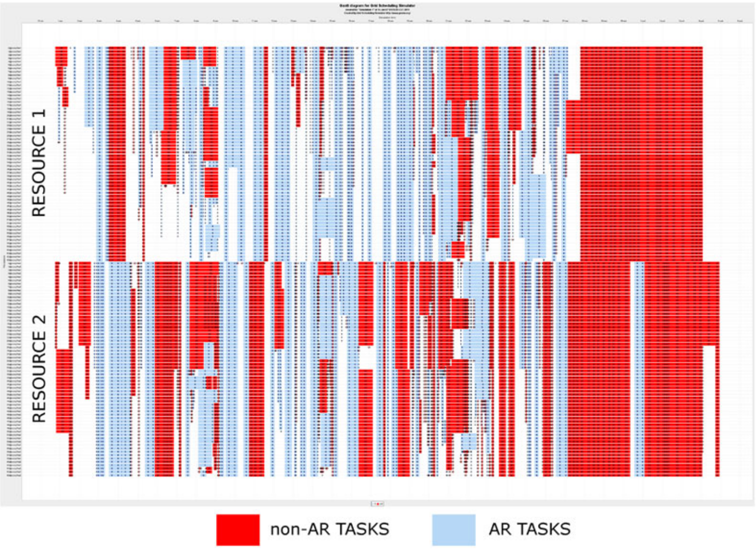


Fig. 7 Gantt chart representation of the schedule including 50 percents of AR tasks and non-AR tasks

Table 4 Average values of the main characteristics of two sets of tasks from the SDSC SP2 workload

Measurement	0–1000	68000–69000
Runtime (s)	8259	8135
Number of requested processors	15	20
Modal value of requested processors	1 (174 times)	64 (147 times)
Requested time (s)	18166	18995
Inter arrival time (s)	1169	2507

Table 5 Evaluation criteria in two workloads for a different number of AR tasks

Percent of *AR*	0	20	40	60	80	100
	mean U					
1–1000	0.54	0.51	0.49	0.45	0.40	0.40
68000–69000	0.83	0.72	0.69	0.68	0.71	0.75
	mean F (s)					
1–1000	29995	31186	30978	31225	27475	22187
68000–69000	48495	90722	101723	110462	98508	78229
	mean D (s)					
1–1000	0	2756	6763	5527	4998	5049
68000–69000	0	4431	3916	4131	17427	34845

this impact will be provided in the following paragraphs of this section.

Additionally, we wanted to compare those two workloads using the *LB* on-line strategy at the grid-level and FCFS in local queues for the increasing number of AR tasks. Table 5 shows obtained results in the light of defined three criteria: *mean F*, *mean U* and *mean D*. Significant differences between those workloads can be related to their characteristics. Among tasks with lower IDs, most of them require only one processor, while in the second set the mode value for this attribute is 64. Further analysis showed another interesting dependence. It turned out that workload 1–1000 contains about 15% of tasks that use 64 processor but their runtime and requested time are 349 and 2041 seconds, respectively. On the other hand, in workload 68000–69000 there is almost 17% of processor intensive tasks with mean runtime equal to 19342 seconds and requested time equal to 37279 seconds. Seeing that impact of AR-tasks on these criteria is more visible for jobs with IDs from 68000 to 69000, we decided to use for the rest of our experiments the second set of tasks taken from the SDSC SP2 workload with the same two-level hierarchical structure.

5.3 The example hierarchical strategies for an increasing number of AR tasks

The scheduling strategy was evaluated according to three different criteria introduced for our new model in Sect. 4, namely *mean U*, *mean F* and *mean D*. It is used for an increasing number of AR tasks performed according to the $ARTS^2$ algorithm using the *CARTS* approach. The processing time limit c_i is usually defined by the local administrator or provided by users during the task submission process. In our experiments this value was read from the workload.

We performed experimental tests of the simplest Random strategy at the grid-level randomly selecting queues for AR and non-AR tasks (FCFS was used at local level). The Random procedure was invoked periodically as the example *off-line* procedure (discussed in Sect. 5.3) according to the off-line scheduling period set at $\delta = 0$, $\delta = 3600$, and $\delta = 86400$ (in seconds). Generated schedules were then evaluated according to the same set of multiple criteria:

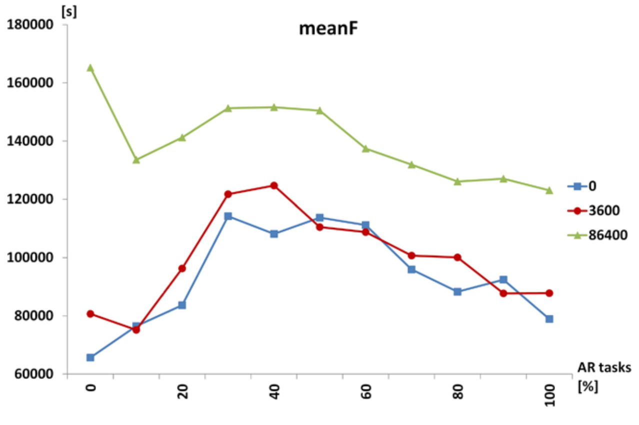


Fig. 8 Flow time related to δ values for a different number of AR tasks

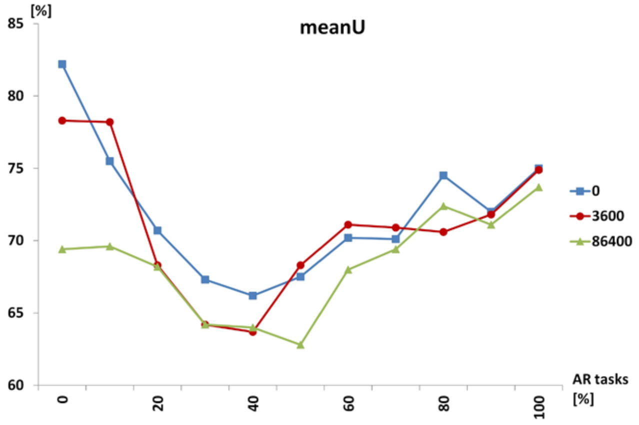


Fig. 9 Resource utilization related to δ values for a different number of AR tasks

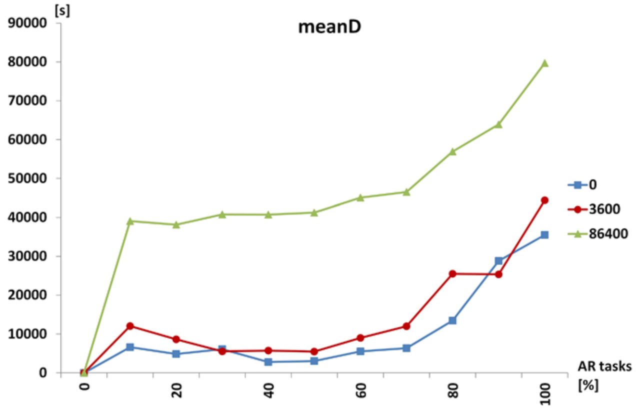


Fig. 10 Tardiness related to δ values for a different number of AR tasks

mean U, *mean F*, and *mean D* as presented in Figs. 8, 9, and 10.

We can easily observe an impact of the increasing offline scheduling period δ on generated schedulers. Naturally, with the increasing value of δ, the overall performance on the *mean F* criteria decreases, especially for a relatively small amount of non-AR tasks. One should note that different δ values do not change performance measured on the *mean U* criterion as tested procedures do not change the order of tasks or optimize their assignment. Nevertheless, if we tested higher δ values, e.g. $\delta = 86400$ we obtained much higher values on the *mean D* criterion (see Fig. 10).

As different computing clusters or supercomputers managed by LRMS may deal with completely different workloads many local administrators enforced rough classification rules to sort tasks into a few classes using a so-called *partitioning* of the system. In fact the partitioning technique gives local administrators the ability to distinguish different tasks during the task submission process. It assigns tasks or resources to various queues, such as: interactive, short, medium, or long queue, according to the expected runtime or specific requirements, in particular advance reservation requests. Therefore, in order to evaluate this approach, we simply divided the original SDSC SP2 resources into two separated queues with 64 computing resources for AR and non-AR tasks, respectively. Obtained results allowed us to analyze how the static partitioning of resources influenced schedules on the considered criteria. With the increasing number of AR tasks we observed better utilization of resources reaching the maximum value for 50% of AR tasks. However, further increase of AR tasks decreases the investigated criterion. On the other hand, as might be expected, the trend for the *mean F* criterion is exactly opposite. Further studies indicated that both functions related to the considered metrics are roughly symmetric about point of equal partitioning of workload into sets with 50% of AR and non-AR tasks. Analysis of the results for the *mean D* criterion showed the slight increase of tardiness until threshold level of about 50% of AR tasks is reached. Then we noticed a rapid deterioration of evaluated measure. It is worth noting that we performed those experiments with different δ values and we have not observed bigger differences on considered criteria. Summarizing, we verified that a static partitioning technique is sensitive to the number of AR tasks in a workload. One should note that the partitioning technique as the administrative policy is not applied often, whereas the workload with a number of AR tasks may change dynamically. Therefore, we decided to improve this approach by applying AR resource priority strategy. This issue is the subject of next section.

5.4 Evaluation of AR resource priority strategy

In this section we present evaluation of the proposed AR resource priority strategy. The major hypothesis to verify is that consolidation of ARs decreases a negative impact of ARs on a mean flow time and resource utilization. Another essential issue to examine is whether the use of the AR resource priority strategy does not deteriorate mean tardiness as this strategy does not attempt to schedule ARs as

Table 6 Comparison of different hierarchical scheduling strategies for 30% of AR tasks

Strategy	*mean U*	*mean F* (s)	*mean D* (s)
Static	0.65	164983	7178
MCT	0.679	107383	5666
AR resource priority	0.695	99714	5576

early as possible. To answer these questions, we compared the AR resource priority strategy with commonly used approach based on the Minimum Completion Time (MCT) algorithm. As we assume homogeneous resources at each resource provider, processing time for each task is identical on each node. Consequently, it is sufficient to allocate AR to a time slot with the earliest start time to implement the MCT strategy. If an AR request contains a fixed interval, available time slots offered by all resource providers have the same start time. If no feasible time slot is found, ARs are searched in larger interval so start times at resource providers may differ. We conducted experiments for two cases: fixed and flexible AR intervals.

5.4.1 Evaluation of AR resource priority strategy for fixed AR intervals

In this experiment we studied impact of the AR resource priority with fixed AR request interval on three evaluation criteria. The obtained results were compared with the MCT algorithm. Table 6 presents the tested strategies and results obtained for three evaluation criteria. In addition to AR resource priority and MCT it also presents results for static configuration of resources—one resource provider for AR tasks and one resource provider for non-AR tasks. Experiments were conducted on the workload with 30% ARs. We have chosen this percentage as higher fraction of AR tasks occur very rarely in existing systems. On the other hand, low numbers of ARs affect the performance in a limited way so they are not of interest in this paper.

As can be observed in Table 6, the AR resource priority strategy significantly outperformed the static strategy. This result was expected as in the case of the static configuration insufficient percentage of resources (50%) was assigned to non-AR tasks. More importantly, AR resource priority turned out to be better than MCT with respect to mean flow time and resource utilization criteria. Thus, these results confirmed our hypothesis concerning positive impact of ARs' consolidation on performance of non-AR tasks and on resource utilization of the whole system. Furthermore, the use of AR resource priority did not deteriorate the mean tardiness, which means that this approach decreases negative impact of ARs on non-AR tasks without worsening performance for user requesting ARs.

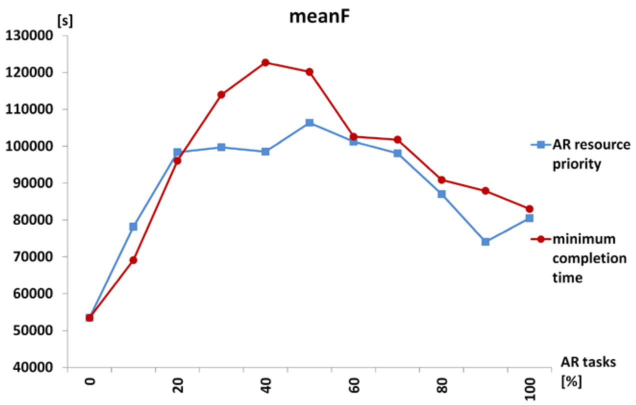


Fig. 11 Flow time for AR resource priority and MCT strategies

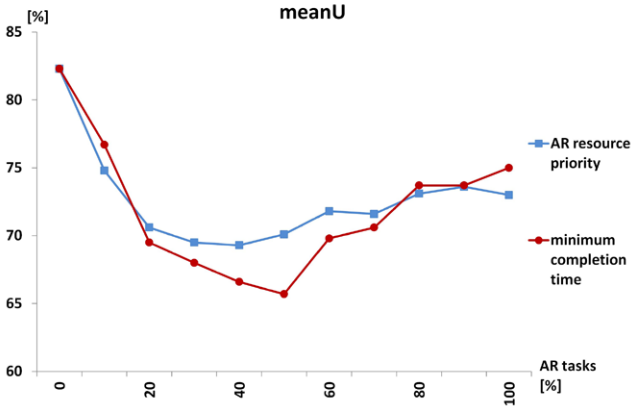


Fig. 12 Resource utilization for AR resource priority and MCT strategies

Figures 11, 12, and 13 show how different numbers of AR tasks affect the *mean U*, *mean F* and *mean D* criteria for both AR resource priority and MCT strategies. Those figures confirm our basic assumption that the increasing number of ARs impacts the quality of schedule. It can be easily concluded that systems that providing only AR capabilities or not supporting them at all perform much better than other. However, one can see again that AR resource priority outperforms MCT with respect to *mean F* and *mean U* criteria. It is worth noticing that the AR resource priority is better than MCT, especially within reasonable ranges of AR tasks, i.e. from 20% to 50%. If a number of AR tasks is very low, their impact on non-AR tasks is negligible. On the other hand, very high percentage of AR tasks is not common in practice. Therefore, good results of a simple AR resource priority strategy in the aforementioned range of AR tasks percentage are very promising.

5.4.2 Evaluation of AR resource priority strategy for flexible AR intervals

In this section we investigate the use of the AR resource priority strategy for flexible AR request intervals. This ex-

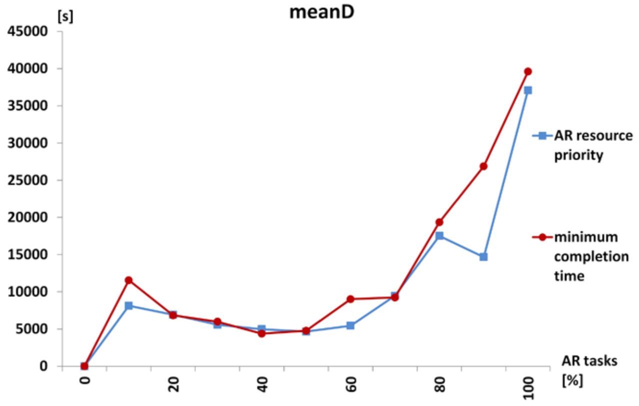


Fig. 13 Tardiness for AR resource priority and MCT strategies

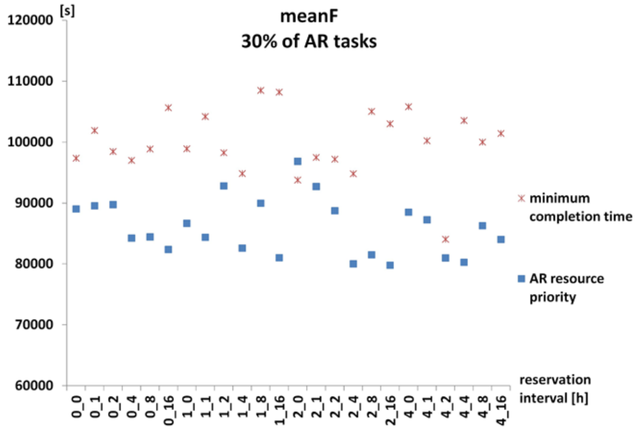


Fig. 14 Flow time for AR resource priority and MCT strategies with respect to length of AR request interval

periment was motivated by a greater flexibility of the MCT strategy in this case and, consequently, a possibility that AR resource priority which does not aim at the earliest start time of AR task, will give substantially worse values of the mean tardiness criterion. However, it turned out that for longer AR request intervals the mean tardiness values were so small that differences between both methods were negligible. Figure 14 shows that AR resource priority gives better results for the majority of AR interval lengths. This experiment was conducted for 30% AR tasks.

Additionally, we studied values of the evaluation criteria for various numbers of AR tasks. The results for interval ranges $[1, 4]$ and $[2, 8]$ are presented in Figs. 15 and 16, respectively. These results showed that AR resource priority policy produces better results with respect to *mean F* and *mean U* also for flexible AR intervals.

To sum up, the AR resource priority strategy reduces the negative impact of ARs in resource management systems without worsening the performance of AR tasks. Furthermore, another advantage of the strategy is its simplicity. As was shown, it can be used as a simple extension of on-line policies at a grid level.

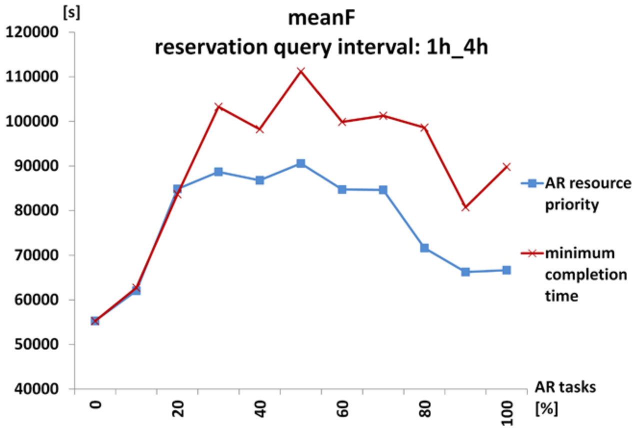


Fig. 15 Flow time for AR resource priority and MCT strategies with AR interval $= [1, 4]$

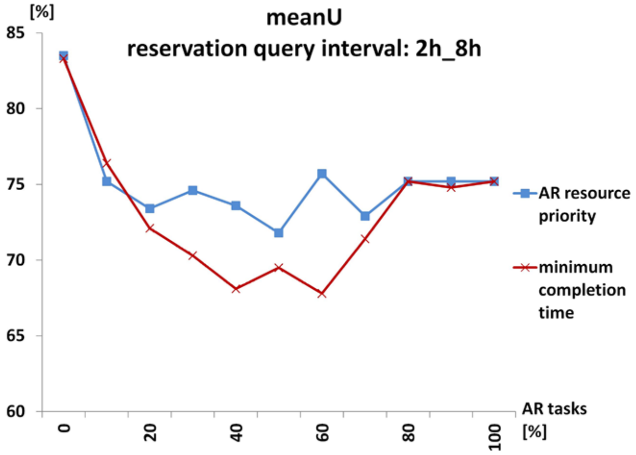


Fig. 16 Resource utilization for AR resource priority and MCT strategies with AR interval $= [2, 8]$

5.5 Finding a trade-off among utilization, mean flow time and tardiness

Three evaluation criteria employed in our experiments represent interests of various groups of stakeholders present in grids. Resource utilization is important for administrators, tardiness for users requesting ARs while flow time is an essential criterion for non-AR tasks. Therefore, in this section we present impact of ARs on three criteria at the same time, and a comparison of AR resource priority with other strategies.

The impact of a number of AR tasks with relation to all tasks on three evaluation criteria is presented in Fig. 17. It is easy to see that increasing a number of AR tasks caused criteria values, especially *mean F* and *mean U* to significantly worsen. After exceeding 50% *mean F* starts again to decrease while *mean U* increase. The reason is that AR tasks become a majority in the system and processing times for them are given.

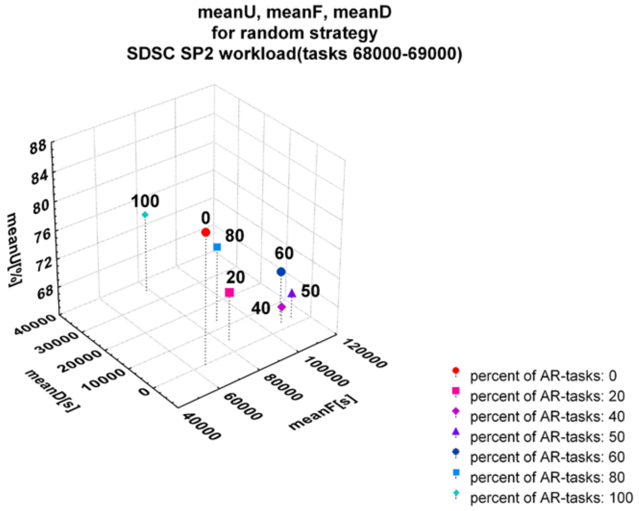


Fig. 17 Evaluation criteria for different number of AR tasks and random strategy

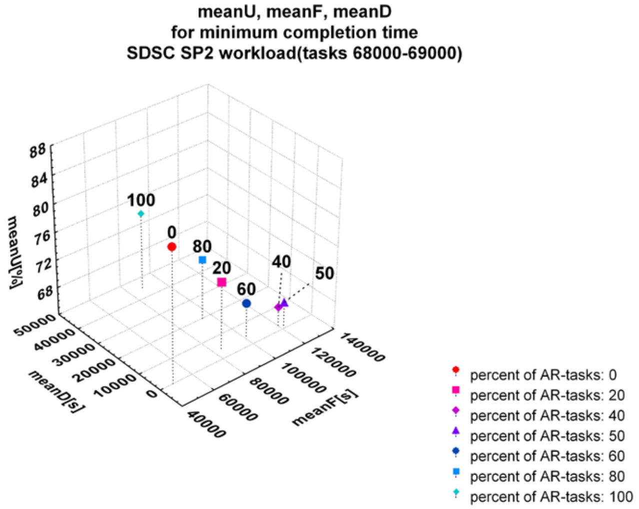


Fig. 18 Evaluation criteria for different number of AR tasks and MCT strategy

The additional two figures allow comparing impact of AR tasks on three criteria for two strategies: AR resource priority and MCT. It is easy to see that AR resource priority does not cause such strong deterioration of *mean F* and *mean U* as it is in the case of MCT when AR tasks number is increasing. It is worth noting that *mean D* changes similarly for both strategies.

Referring again to the comparison Table 6, one can see that the AR resource priority offers better solution than MCT strategy with respect to mean flow time and resource utilization without worsening mean tardiness criterion. In this way, it confirms the proposed direction of consolidating AR tasks together and, consequently, obtaining better schedules in grids with mixed AR and non-AR tasks.

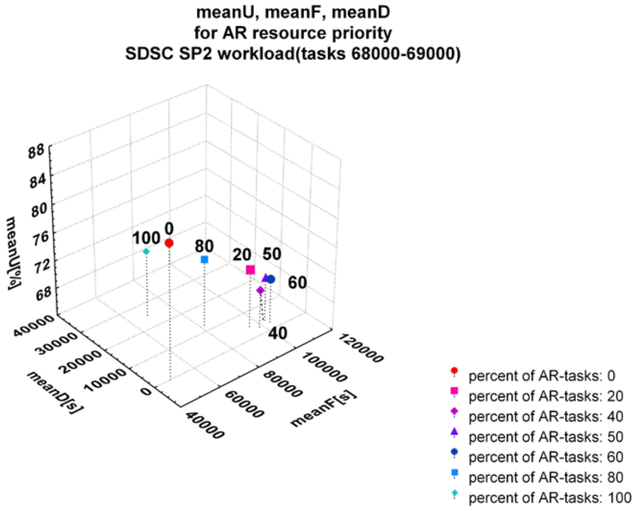


Fig. 19 Evaluation criteria for different number of AR tasks and AR resource priority strategy

Promising results of the AR resource priority strategy with respect to the mean tardiness criterion are caused by conservative approach of this strategy. It selects a resource provider with a higher priority but only if feasible AR task start time is within the requested AR interval. Therefore start times provided by the AR Resource Priority and MCT algorithms are similar especially in the fixed AR interval case. It is worth noting that the AR Resource Priority strategy may negatively affect the mean flow time of AR tasks, especially with the flexible AR intervals. In larger intervals MCT may find substantially earlier start times. Nevertheless, we did not studied this in detail as we assumed that the tardiness is a major criterion significant for AR tasks' users. Relation of mean flow time of non-AR and AR tasks for various scheduling strategies is an interesting topic for future research.

6 Conclusions

Recently, advance reservations (ARs) have been gaining higher importance in some systems due to increasing popularity of modern applications that require interactive tasks, co-allocation of multiple resources, and performance guarantees. However, simultaneous scheduling of both AR and non-AR tasks is not easy as ARs significantly deteriorate flow time of batch tasks and the overall resource utilization. In this paper we studied mutual impact of non-AR and AR tasks and evaluated various strategies that can be easily used for efficient hierarchical scheduling in grids. First, we investigated scheduling of batch tasks and reservations separately and presented relevant insights. We demonstrated for non-AR tasks how basic on-line scheduling policies can be outperformed by a relatively simple off-line scheduling policy such as LSF if an appropriate invocation period δ is applied. In general, according to the obtained results, short δ periods do not have a huge impact on the overall performance in the light of considered criteria. Moreover, it has a negative impact on performance for mixed AR and non-AR tasks in case of simple online scheduling policies. On the other hand, the off-line scheduling period gives a great opportunity to apply more sophisticated scheduling optimization procedures at both grid and local levels. As many evaluation metrics should be considered at the same time during the optimization process there is a place for new multi-criteria and adaptive approaches. We believe that there is always a trade-off between on-line and off-line configuration parameters, in particular δ, which in practice should be adjusted experimentally as different real workloads may have completely different characteristics.

As the use of ARs significantly impacts the resource utilization what was demonstrated in Sect. 3, we proposed a new data structure based on variable-length slots which improved the utilization with performance comparable to common fixed-slot AR approaches. We have shown that dynamic binding of computing resources (nodes) to AR (i.e. at its start time) is a sufficient condition to determine about feasibility of AR at a given time based only on a number of free nodes (without checking specific nodes). Dynamic binding is important not only because it allows a grid-level scheduler to easily and efficiently check the availability of underlying resources for AR, but it also gives more flexibility to a local-level scheduler which can allocate AR tasks more efficiently without blocking non-AR tasks. Therefore, we applied dynamic binding in all experiments we performed with the use of ARs.

During our experiments we discovered that the main reason of worse flow time and resource utilization are ARs allocated just after running batch tasks. Due to lack of knowledge of processing times for non-AR tasks, much longer runtime limit times than actual task runtimes were used for scheduling. We have confirmed that for a mixed workload consisting of batch tasks and AR tasks it is worth to group together AR requests. In other words, if possible, ARs should be allocated after another AR rather than after a batch task. We proposed a new hierarchical scheduling strategy combining features of LB and FCFS with backfilling which dynamically groups ARs and does not require a static partitioning of computing resources. Moreover, according to the obtained results, it has generated better schedules on all three relevant criteria, namely *mean* U, *mean* F and *mean* D, and for a different number of AR tasks. In general, to reduce a negative impact of ARs and batch tasks on each other the following basic rules should be taken into account:

- Use as much flexibility as possible: (i) dynamic binding of ARs to avoid static blocking of specific nodes for batch tasks and (ii) flexible, rather than fixed, AR requested start and end time (intervals)
- Group ARs together to avoid setting reservations after running batch tasks. For instance, set resource priorities that bias decisions about allocation of advance reservations
- Perform experiments with a given real workload to find and apply the best δ period for off-line scheduling strategies

One should note that the proposed evaluation criteria that satisfy administrators, users of batch tasks, and users who establish reservations are contradictory. Often, improvement of one criterion may lead to deterioration of another. Therefore, it is possible to find diverse Pareto-optimal configurations of methods, however, the selection of a single compromise solution may require preferential information and adoption of one of the multi-criteria methods, e.g. a new multi-criteria job scheduling method to find a fair schedule of jobs that were submitted by multiple users introduced in Kurowski et al. (2010).

In this paper we clearly indicated that results heavily depend on the particular workload, steering scheduling parameters and configuration of resources. In particular, large tasks and large ARs make the adoption of suggestions presented above difficult. Therefore, we performed most experiments on the real and well-known workload. Nevertheless, tests on other workloads with different characteristics and synthetic workloads may give more insights concerning dependence of workload parameters on efficiency of hierarchical scheduling methods. These tests are one of goals of our future work. Another future work includes off-line optimization (in batches) using various heuristics that should lead to better schedules for mixed AR and non-AR tasks. As we consider three different evaluation criteria to satisfy administrators, batch task users, and users who require advance reservations, multi-criteria optimization methods should be applied. We also would like to further study the consolidation of ARs and apply it on a local level and in optimization methods.

Acknowledgements The SDSC SP2 workload used during our experiments was brought by the HPC Systems group of the San Diego Supercomputer Center (SDSC). The JOBLOG data is Copyright 2000 The Regents of the University of California All Rights Reserved.

References

Aida, K., & Casanova, H. (2009). Scheduling mixed-parallel applications with advance reservations. *Cluster Computing*, *12*(2), 205–220.

Brodnik, A., & Nilsson, A. (2003). A static data structure for discrete advance bandwidth reservations on the Internet. Tech report, cs.DS/0308041. http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0308041.

Burchard, L.-O. (2005). Analysis of data structures for admission control of advance reservation requests. *IEEE Transactions on Knowledge and Data Engineering*, *17*(3), 413–424.

Feitelson, D. G. (2008). Looking at data. In *Proceedings 22nd intl. parallel & distributed processing symp. (IPDPS)*

Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., & Epema, D. H. J. (2008). The grid workloads archive. *Future Generations Computer Systems*, *24*(7), 672–686.

Jones, J.P., Nitzberg, B. (1999). Scheduling for parallel supercomputing: a historical perspective on achievable utilization. In *Lecture notes in computer science: Vol. 1659. Proceedings of the workshop on job scheduling strategies for parallel processing (JSSPP)* Berlin: Springer.

Kurowski, K., Nabrzyski, J., Oleksiak, A., & Weglarz, J. (2006a). Scheduling jobs on the grid—multicriteria approach. *Computational Methods in Science and Technology*, *12*(2), 122–138.

Kurowski, K., Nabrzyski, J., Oleksiak, A., & Weglarz, J. (2006b). Grid multicriteria job scheduling with resource reservation and prediction mechanisms. In J. Józefowska & J. Weglarz (Eds.), *Perspectives in modern project scheduling* (pp. 345–373). New York: Springer.

Kurowski, K., Nabrzyski, J., Oleksiak, A., & Weglarz, J. (2007). Grid scheduling simulations with GSSIM. In *Proceedings of the international conference on parallel and distributed systems*. doi:10.1109/ICPADS.2007.4447835.

Kurowski, K., Nabrzyski, J., Oleksiak, A., & Weglarz, J. (2008). Multicriteria approach to two-level hierarchy scheduling in grids. *Journal of Scheduling*, *11*(5), 371–379.

Kurowski, K., Oleksiak, A., & Weglarz, J. (2010). Multicriteria, multi-user scheduling in grids with advance reservation. *Journal of Scheduling*, *13*(5), 493–508.

Maui (2010). http://www.clusterresources.com/products/maui/.

Oleksiak, A. (2009). Multicriteria job scheduling in grids using prediction and advance resource reservation mechanisms. PhD thesis, Poznan University of Technology.

Parallel Workload Archive (1999). Chapin, S. J., Cirne, W., Feitelson, D. G., Jones, J. P., Leutenegger, S. T., Schwiegelshohn, U., Smith, W. & Talby, D. Benchmarks and standards for the evaluation of parallel job schedulers. In *Job scheduling strategies for parallel processing*, Feitelson, D. G. & Rudolph, L. (Eds.).

Parallel Workload Archive (2010). http://www.cs.huji.ac.il/labs/parallel/workload/.

PBS (2010). http://www.pbsgridworks.com/.

Platform LSF (2010) http://www.platform.com/.

SGE (2010). http://wikis.sun.com/display/GridEngine/Home.

SMOA Computing (2010). http://www.qoscosgrid.org/trac/qcg-computing.

Sulistio, A., Kim, K. H., & Buyya, R. (2007). On incorporating an on-line strip packing algorithm into elastic grid reservation-based systems. In *Proceedings of the 13th international conference on parallel and distributed systems*, ICPADS, 2007. doi:10.1109/ICPADS.2007.4447738.

Sulistio, A., Cibej, U., Prasad, S., & Buyya, R. (2008). GarQ: an efficient scheduling data structure for advance reservations of grid resources. *International Journal of Parallel, Emergent and Distributed Systems*, doi:10.1080/17445760801988979.

Tchernykh, A., Ramarez, J., Avetisyan, A., Kuzjurin, N., Grushin, D., & Zhuk, S. (2006). Two-level job-scheduling strategies for a computational grid. In X. Wyrzykowski, et al. (Ed.), *LNCS: Vol. 3911. Parallel processing and applied mathematics, the second grid resource management workshop (GRMW'2005) in conjunction with the sixth international conference on parallel processing and applied mathematics—PPAM 2005. Poznan* (pp. 774–781). Berlin: Springer.

Xiong, Q., Wu, C., Xing, J., Wu, L., & Zhang, H. (2005). A linked-list data structure for advance reservation admission control. In *Networking and mobile computing* (pp. 901–910). Berlin: Springer.