

Elastic Grid Reservations with User-Defined Optimization Policies

Thomas Röblitz, Florian Schintke, Jan Wendler
Zuse Institute Berlin (ZIB)
Takustr. 7, D-14195 Berlin, Germany
{roebnitz,schintke,wendler}@zib.de

Abstract

We present an algorithm for reserving compute resources in a Grid, that allows users to define an optimization strategy, if multiple candidates match the specified requirements. Reservations need not to be exact, but can be elastic in some properties like start-, end-time, duration or the number of requested CPUs. Our algorithm takes the speedup of applications into account to accordingly adjust the duration of the reservation when changing the number of CPUs.

This new reservation approach allows users to more accurately specify their requirements and gives the computer system more flexibility to find appropriate resources than existing solutions. Caused by the enhanced flexibility, a single request can have many solutions to select from. To perform this selection efficiently in both cases, successful and failed requests, was one of our goals and we show performance results in Section 4.

This work can also build as a basic block for flexible co-reservations – the allocation of multiple independent resources. Due to the elasticity of our reservation requests, the probability to find a common time-slot on all requested resources is higher with elastic requests than with fixed requests.

1. Introduction

Job execution in Grid environments is dominated by the classical batch-processing, where a broker chooses a site where the job is sent to and then the local scheduler decides when to execute the job. The Grid broker has no influence on the actual start time of the job, which is sufficient for usual high-throughput computing.

To provide better guarantees, the broker can try to estimate the actual start time by looking at parameters like the queue length. Despite queue wait time estimation techniques [10], such estimations are still very inaccurate in many workload situations. Especially for live demonstrations, coordinated usage of multiple resources, or the processing of a certain workload before a given deadline, bet-

ter guarantees are needed. Reservations are a means to provide such higher guarantees.

Today, reservation requests are typically 'exact' in terms of number of processors, network bandwidth, start- and end-time. We call these requests *rigid*, because there is no freedom to 'negotiate' some parameters when the exact request fails (shifting it five minutes could be enough). Additionally, such requests are usually sent directly to the specific resource management systems instead via a middleware. This makes reservation requests often incompatible between different resource management systems, because they are expressed in different languages.

Systems could further benefit from elastic reservation requests since the advent of 'metacomputing' [8], where several independent batch systems are used concurrently. Therefore several sub-jobs are queued to different batch systems. The sub-jobs are started uncoordinated, but the complete job can only begin, if all sub-jobs are running. In between, already started sub-jobs are idle and waste computing resources. While co-reservations solve this problem, elastic reservations help to automatically find common time-slots that are available at all involved batch systems.

Methods for reserving resources in networks and multimedia environments have been studied extensively, e.g. [13] and [12]. The latter introduces a probing mechanism to efficiently determine available network bandwidth. We adopted this method for CPU resources in our algorithms.

In [11] and [9], book-ahead reservation methods were studied with respect to their impact on the scheduling of user jobs in the system providing the resources. Our methods introduce more flexibility in requesting reservations, i.e. not only in the start time.

In Grid environments, previous work on advance reservations has concentrated on the overall applicability to the allocation of resources for job management [4]. In this paper we present an efficient algorithm that processes *elastic* reservation requests for CPU resources. Our reservation requests are elastic in the number and type of processors, the earliest start and the latest end time and the duration of the reservation.

2. Architectural Overview

In this section we present the parameters an elastic reservation request consists of and the components of a reservation framework which processes these requests.

2.1. Parameters of an Elastic Reservation Request

For an elastic reservation, users can specify an execution window by defining the earliest start time (est) and the latest end time (let), the duration (dur_{ref}) and the number of CPUs (np_{ref}) for the reservation. To allow elasticity in the processor type a reference duration (dur_{ref}) is defined for a specific power (pp_{ref}) and number (np_{ref}) of processors. The algorithm adjusts the requested duration to the actual processor type and number by scaling it according to the speedup (sp), which is defined using speed-up models such as *Amdahl* [1], *Downey* [2], or using a database with several reference values. Users additionally define the range of CPUs (np_{min} , np_{max}) their application can run with.

Name	Description	Example
ucp	user's certificate proxy	XerTWQ4
est	earliest start time	06:00
let	latest end time	20:00
np_{min}	min. # of processors	2
np_{max}	max. # of processors	32
dur_{ref}	reference duration	2h
pp_{ref}	reference processing power	SPECint2000:1500
np_{ref}	reference # of processors	8
sp	speedup as: model/param	amdahl/0.01
pref	set of prioritized preferences	1. end time, 2. cost

Table 1. Parameters of an elastic reservation request.

Users also can specify an optimization policy ($pref$) according to their goals. There can be many eligible *reservation candidates* where the algorithm chooses the best in respect to the user's optimization policy. An optimization policy is a list of selection criteria ordered by decreasing importance. If a user wants to make a demonstration during a class, the execution window and the duration would have highest priority. But if a user wants to run a large and possibly expensive job, low costs for the execution may have the highest priority.

A complete user's reservation request is defined by the parameters shown in Table 1.

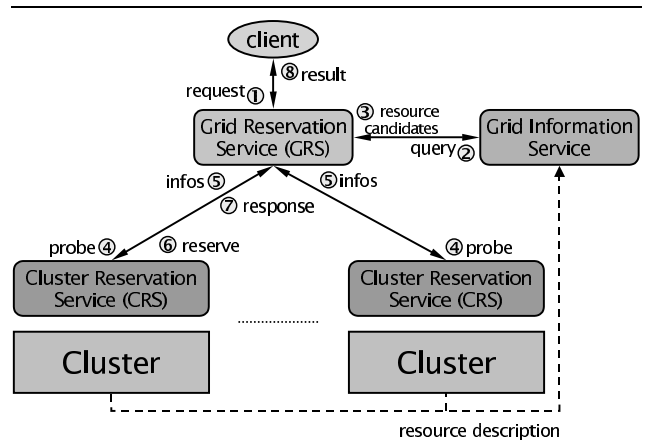


Figure 1. Components of the reservation framework and their interplay for an elastic reservation request.

2.2. Components of the Reservation Framework

A reservation request is processed by different components in a Grid environment. Fig. 1 shows the involved services and their interactions.

Each cluster is extended by a *Cluster Reservation Service* (CRS). The CRS provides a standard interface to the *Grid Reservation Service* (GRS) and may enhance the functionality of the underlying cluster batch system, i.e. for obtaining detailed utilization information or enabling reservation facilities [7]. A Grid environment may run multiple GRS instances, e.g. per user, per virtual organization (VO), etc. The GRS is the interface to clients and performs the core of the algorithm.

A flexible reservation request is processed with the following 8 steps (compare Fig. 1):

- ① A client sends a request to a GRS instance.
- ② The GRS instance queries a *Grid Information Service* (GIS) for resources which satisfy the core requirements, e.g. OS type, access rules, etc.
- ③ The GIS returns a set of *resource candidates* (RC).
- ④ The GRS sends probe requests to the CRSs of the RCs to obtain information about their resource availability.
- ⑤ The contacted CRS instances send back a list of time slots. Each time slot contains a set of attributes, e.g. its start and end time, number of CPUs, cost, estimated success rate, etc.
- ⑥ (a) The GRS orders all retrieved time slots according to the defined optimization policy (user preferences).
(b) The GRS tries to reserve the best reservation candidate (time slot at a resource candidate) *or* continues with step ⑧ if the list of eligible time slots is empty.
- ⑦ The GRS receives the result of the reserve request. If the request failed, it removes the reservation can-

didate and continues with step ⑥b. Otherwise it continues with step ⑧.

- ⑧ The GRS sends a response to the client. If the resource has been reserved, the response includes the negotiated terms, e.g. number of processors, start time, end time, cluster reservation ID, etc.

The processing consists of three major phases: the *GIS phase* for finding resources (steps 2 and 3), the *Probe phase* for obtaining detailed status information about the resources (steps 4 and 5) and the *Reserve phase* to actually reserve processors at a cluster (steps 6 and 7).

3. Reservation Algorithm

For the design of our reservation algorithm the following properties are of main importance:

Efficiency: the less overhead is needed to process a successful or failed request the better. The efficiency can be measured in terms of time or number of exchanged messages. There is a tradeoff between the degree of flexibility (size of the parameter space) and the efficiency.

Impact on user jobs: normal batch jobs should be delayed as less as possible by reservations. Reservations not only reduce the number of processors available for batch jobs, but also reduce the optimization possibilities for the scheduler. Hence, batch jobs may wait longer before they can start.

Satisfaction: describes how well the user's preferences are met. There is a tradeoff between the satisfaction and the efficiency as well as the impact on user jobs.

In the following sections, we describe in detail our algorithm along the processing phases of a request (1st *GIS*, 2nd *Probe*, 3rd *Reserve*).

3.1. GIS Phase

After receiving the request, the GRS determines available sites by querying a *Grid Information Service*. Available sites are those that satisfy the core requirements, e.g. the minimally required number of processors, the requested type of operating system, etc. and provide reservation capabilities.

3.2. Probe Phase

In [12], a probing mechanism is introduced to exploit the potential flexibility in reservations for distributed multimedia applications. Our approach is similar with respect to the concept of obtaining future utilization information.

Because, a site's processor power may differ from the reference value pp_{ref} (a parameter of a reservation request),

the GRS must adjust the duration and processor range for each site before sending a probe request (details in Section 3.2.1).

After adjusting the request parameters, the GRS sends a probe request to the CRS of each site (details in Section 3.2.2). Each CRS determines a set of start times (or time slots), calculates the requested properties and sends the collected information back to the GRS (details in Section 3.2.3).

3.2.1. Adapting the Request Parameters. The elastic reservation request specifies the duration (dur_{ref}) for a reference system including the number of processors (np_{ref}) and an abstract processor power (pp_{ref}), e.g. SPEC CPU2000. Assuming the processor power of a site is given by pp_{site} , the new duration is determined as follows¹

$$dur_{new} := dur_{ref} \cdot \frac{pp_{ref}}{pp_{site}}$$

We define the maximum duration as $dur_{max} := let - est$. If $dur_{new} > dur_{max}$ holds, the lower end of the range of processors (np_{min}) may be increased to obtain feasible durations ($dur_{new} \leq dur_{max}$).

Based on speedup models the minimally required number of processors can be derived from the relation

$$\frac{dur_{new}}{dur_{max}} = \frac{S_{A/D}(np_{min}^{new})}{S_{A/D}(np_{ref})} \quad (1)$$

The speedup functions S_A (Amdahl's model [1]) and S_D (Downey's model [2]) are defined as follows.

$$S_A(n) := \frac{seq + par}{seq + \frac{par}{n}}, \quad (2)$$

where $n \in \mathcal{N}$ is the number of processors, seq is the sequential and par is the parallel fraction of the program. Thus, $seq + par = 1$ holds for any program.

$$S_D(n) := \begin{cases} \frac{An}{A+\sigma/2(n-1)} & \sigma \leq 1 \wedge 1 \leq n \leq A \\ \frac{An}{\sigma(A-1/2)+n(1-\sigma/2)} & \sigma \leq 1 \wedge A \leq n \leq 2A-1 \\ A & \sigma \leq 1 \wedge n \geq 2A-1 \\ \frac{An(\sigma+1)}{\sigma(n+A-1)+A} & \sigma \geq 1 \wedge 1 \leq n \leq A+A\sigma-\sigma \\ A & \sigma \geq 1 \wedge n \geq A+A\sigma-\sigma \end{cases} \quad (3)$$

where $n \in \mathcal{N}$ is the number of processors, A is the average parallelism of the program and σ its variance of parallelism.

Based on the equation (1) we describe the calculation of the adapted minimum number of processors for each speedup model in detail. Let np_{site} be the maximum number of processors of a site.

¹ This simplification may not always be true. Especially, if a program's runtime not only depends on the processor performance, but also on other important components, e.g. network, disk, etc.

```

funct  $np\_min\_adapt(np_{min}^{new}, np_{max}, C, np_{site}) \equiv$ 
  while  $C > S_D(np_{min}^{new}) \wedge np_{min}^{new} \leq np_{max}$  do
     $np_{min}^{new} := np_{min}^{new} + 1;$ 
  od
  if  $C < S_D(np_{min}^{new})$ 
    then
       $np_{min}^{new} := \max(np_{site}, np_{max}) + 1$ 
    fi.

```

Figure 2. The calculation of the lower bound of the processor range based on Downey's speedup model.

Amdahl's Model. Based on the definition (2) and equation (1) we derive

$$np_{min}^{new} := \left\lceil \frac{1-s}{\frac{dur_{max}}{S_A(np_{ref}) \cdot dur_{new}} - s} \right\rceil \quad (4)$$

where seq is the sequential part of the program (part of request parameter sp). For $0 < seq < 1$ the condition

$$\frac{1}{S_A(np_{ref}) \cdot seq} > \frac{dur_{new}}{dur_{max}}$$

must hold or the resource candidate is not valid. If $seq = 0$, equation (4) can be simplified to

$$np_{min}^{new} := \left\lceil \frac{S_A(np_{ref}) \cdot dur_{new}}{dur_{max}} \right\rceil$$

If $seq = 1$, no speedup is possible. Thus, the resource candidate is not valid.

Downey's Model. Due to the nature of definition (3), the lower bound of the new processor range np_{min}^{new} is determined in an iterative procedure as shown in Fig. 2. The constant C (required speedup) is defined as

$$C := S_D(np_{ref}) \cdot \frac{dur_{new}}{dur_{max}}$$

If the newly calculated lower end of the range is larger than np_{max} or np_{site} , the resource candidate is not valid.

3.2.2. Sending a Probe Request. Most of the attributes of a probe request are taken over from an elastic reservation request (compare with the set of attributes in Table 1). These attributes are: ucp , est , let , np_{max} and sp .

The values of the attributes np_{min} , dur_{ref} and np_{ref} are adapted as described in Section 3.2.1. If the value of np_{min} has been changed, the value of np_{ref} is set to the new value as well.

Additionally, a probe request contains the attributes $prop$, tsn_{max} and tss_{gap} . The attribute $prop$ contains a list of properties to be determined for each time slot. This list is generated from the set of user preferences $pref$. The number of

```

funct  $process\_probe(see\ Section\ 3.2.2) \equiv$ 
   $RSVC := \{\};$ 
   $\forall n \in RNP$ 
  do  $dur_n := dur\_adapt(n, dur_{ref}, np_{ref}, sp);$ 
    if  $dur_n \leq dur_{max}$ 
      then
         $STT_n := start\_times(est, let, tsn_{max}, tss_{gap}, dur_n);$ 
         $\forall t \in STT_n$ 
        do  $ps := get\_properties(t, dur_n, n, prop);$ 
           $append(RSVC, \langle site, t, dur_n, n, ps \rangle);$ 
        od
      fi od.

```

Figure 3. The main procedure of the probe request processing performed by the Cluster Reservation Service.

time slots to be generated is limited by the attribute tsn_{max} . The attribute tss_{gap} defines a lower limit for the gap between the start times of two consecutive time slots (compare Fig. 4).

3.2.3. Processing a Probe Request. The Cluster Reservation Service handles the probe request. Fig. 3 shows the main procedure of the CRS. RNP is the set of reservable numbers of processors, i.e. some cluster management systems do not assign individual processors to requests, but bunches of processors.

The adaptation of the duration ($dur_adapt()$) is defined as

$$dur_adapt(n, dur_{ref}, np_{ref}, sp) := dur_{ref} * \frac{S_{A/D}(np_{ref})}{S_{A/D}(n)} \quad (5)$$

where $n \in \mathcal{X}$, $np_{min} \leq n \leq np_{max}$ and $S_{A/D}$ as defined in (2) and (3).

The function $start_times()$ calculates the set STT_n of feasible start times of time slots for a specific number of processors. Fig. 4 shows a distribution of time slots over the requested period of time $[est, let]$ for different number of processors. The time between the start times of two consecutive time slots is the *gap* between these time slots. The latest start time lst_n for a given number of processors n is defined as $lst_n := let - dur_n$, where dur_n is the adapted duration.

The distribution of time slots must fulfill the following conditions:

- the requested period of time $[est, let]$ should be well covered,
- the number of time slots is limited by the parameter tsn_{max} and
- the gap between two consecutive time slots should be larger than or equal to the parameter tss_{gap} .

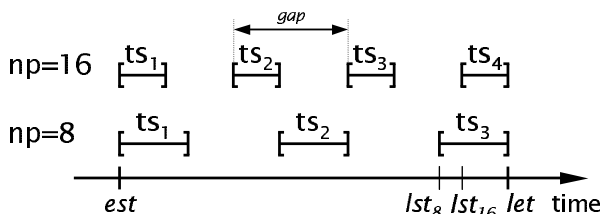


Figure 4. Distribution of time slots over the requested period of time $[est, let]$.

Before the CRS appends a reservation candidate tuple

$$rsvc := \langle site, start, end, np, ps \rangle$$

to the list of results $RSVC$, it calculates the requested properties. Here, we discuss methods to calculate the properties cost and esr. These are stored in the set of properties ps .

The cost of a reservation candidate. Each resource provider defines a base unit cost BU_c . A base unit has a duration BU_d and is applicable for a single processor. Because the demand for resources may change over the day, we define scalings of the base unit cost, e.g. a factor of 0.5 may be applied for nightly hours. Also, the resource provider may have different agreements with different users or virtual organizations. Hence, the overall cost for a reservation candidate $rsvc$ can be calculated as follows:

$$cost(rsvc, ucp) := np \frac{BU_c}{BU_d} \int_{start}^{end} scale(t, ucp) dt \quad (6)$$

where $scale()$ calculates the scaling with respect to time t and agreements with the user ucp .

The estimated reservation success rate of a reservation candidate. Whether or not a reservation candidate can be successfully reserved depends on many factors, e.g. current utilization of a site, size of the request, scheduling policies, etc. The property esr (estimated success rate) provides a mean to abstract from all these factors and to express the probability that a reservation candidate may be successful. Here we propose the methods *static*, *history* and *load* to calculate a value for the property $esr \in [0, 1]$.

The method *static* calculates the esr based on the distance d from the current time ct until the beginning (*start*) of the reservation candidate. According to [5] for reserving network bandwidth, there is a certain threshold for the book-ahead time. Before this threshold very few to none requests are granted. After the threshold almost all requests are granted. The corresponding function esr_S is defined as follows

$$esr_S(d) := 1 - e^{-\frac{d}{h}}$$

where h is a configurable constant of a CRS. It can be used to adjust the curve of the esr function.

The method *history* is based on recorded utilization information, i.e. number of idle processors. For a time interval $[begin, end)$ the number of idle processors is taken at time *begin* and recorded in the history set H . All intervals have the same length δ . A recorded value for time t is accessed with the function $H_{idle}()$, which returns the value for the interval to which time t belongs to ($begin \leq t < end$).

Given a reservation candidate with the time interval $[s, e]$ we determine the average number of idle processors $aip()$ in the system using a daily pattern $dp()$ with the formula

$$aip(s, e, H) := \frac{1}{\sum_{t \in H} dp(t, s, e)} \sum_{t \in H} H_{idle}(t) \cdot dp(t, s, e) dt$$

The pattern function $dp(t, s, e)$ determines if the day time of a recorded interval intersects with the day time of the reservation candidate. If so it is set to 1, otherwise to zero.

The function esr_H is defined as follows

$$esr_H(s, e, np, H) := \begin{cases} 1 & , \text{if } 2np \leq aip(s, e, H) \\ 2 - \frac{2np}{aip(s, e, H)} & , \text{if } np \leq aip(s, e, H) \\ 0 & , \text{else} \end{cases}$$

The method *load* uses currently available information on running and waiting jobs and existing reservations. With this information the approximative end time T_{wkl} for the known workload is calculated. Given a reservation candidate with the start time s , we define the function esr_L as follows

$$esr_L(s, T_{wkl}) := \begin{cases} 1 & , \text{if } s \geq T_{wkl} \\ 0 & , \text{if } s < T_{wkl} \end{cases}$$

The approximative workload end time T_{wkl} is calculated as follows. Let J_r be the set of running jobs which do not use reserved processors. For each $j \in J_r$, j_{ret} denotes the remaining execution time, whereas j_{np} denotes the number of used processors. The parameter acc_r specifies the accuracy of the remaining execution time. Let J_w be the set of waiting jobs. For each $j \in J_w$, j_{wt} denotes the wall time limit, whereas j_{np} denotes the number of requested processors. The parameter acc_w specifies the accuracy of the wall time limit. The set RSV holds the existing reservations. For each $r \in RSV$, r_s , r_e and r_{np} denote the start time, the end time and the number of processors of the reservation, respectively.

The approximation is done in three steps. First, we determine the average end time of the running jobs.

$$T_{J_r} := \frac{1}{\sum_{j \in J_r} j_{np}} \sum_{j \in J_r} j_{np} \cdot j_{ret} \cdot acc_r$$

Second, we calculate the average end time for the waiting jobs using their wall time limit.

$$T_{J_w} := \frac{1}{np_{site}} \sum_{j \in J_w} j_{np} \cdot j_{wt} \cdot acc_w$$

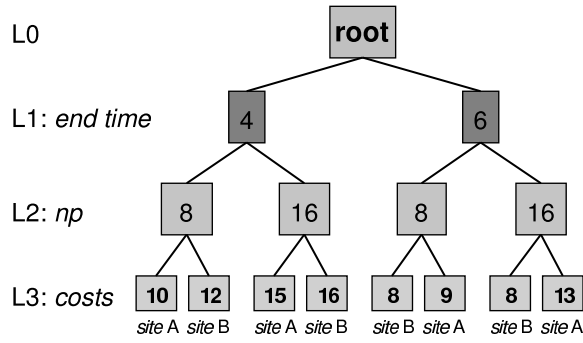


Figure 5. Sorting reservation candidates in a tree structure. The left most leaf represents the first candidate to reserve.

The temporary T_{wkl} is set to the sum of T_{J_r} and T_{J_w} . Third, we take existing reservations (which may be active between the current time ct and T_{wkl}) into account by iteratively increasing T_{wkl} for such reservations. Clearly, the value for T_{wkl} is only a rough approximation, but it should be reasonable in many workload situations, especially if good accuracy parameters were used.

3.3. Reserve Phase

The task of the *Reserve phase* is to find a reservation candidate which can be successfully reserved. Because the number of reservation candidates may be very large, we use filtering and sorting to try the ones that promise a high success rate and high values for the objective functions (user preferences).

3.3.1. Filtering and Sorting the Reservation Candidates. Considering a set of reservation candidates $RSVC = \{\langle site, s, e, np, ps \rangle\}$ where the list of properties ps contains at least the attribute esr (estimated success rate). The filtering is done by considering only those candidates for which the constraint $esr \geq th$ holds. The threshold th is a constant of the *Grid Reservation Service* or can be specified for each reservation request individually.

The remaining candidates RRC are sorted according to the user defined objective functions (user preferences). We use a tree structure, where the root (level L0) represents the set RRC . A node at level L_i represents reservation candidates for which the objective functions with the priorities P_1, \dots, P_i evaluate to the same value. The children of a node at level L_i are sorted according to the objective function with priority $i + 1$. Hence, the tree has a maximum depth of m . The leafs may contain sets of reservation candidates, which have the same values of the objective functions for all priorities. Fig. 5

shows an example with 8 remaining reservation candidates (format is $\langle site, start, end, np, costs \rangle$): $\langle A, 0, 4, 8, 10 \rangle$, $\langle A, 1, 4, 16, 15 \rangle$, $\langle A, 2, 6, 8, 9 \rangle$, $\langle A, 3, 6, 16, 13 \rangle$, $\langle B, 1, 4, 8, 12 \rangle$, $\langle B, 2, 4, 16, 16 \rangle$, $\langle B, 3, 6, 8, 8 \rangle$, $\langle B, 4, 6, 16, 8 \rangle$. The objective functions are $p_1(rsvc) = rsvc.end$, $p_2(rsvc) = rsvc.np$ and $p_3(rsvc) = rsvc.costs$. The attributes $site$ and $start$ are not directly referenced in the objective functions. However, since the $costs$ depends on the $site$ and time slot $[start, end]$ these attributes do have influence on the ordering.

3.3.2. Reserving. The resulting tree is traversed in *depth-first* order. For each leaf a reservation request is sent to the corresponding $site$. The GRS continues until a request succeeds or all reservation candidates have been tried to reserve. If a CRS receives a reserve request, it verifies whether the known workload (running and waiting jobs and existing reservations) conflicts with the requests. This check uses the calculation of T_{wkl} as described in Section 3.2.3 (paragraph on calculating esr_L). If the requested time interval begins after T_{wkl} , the request is passed to the local scheduler. Otherwise it is rejected. This check ensures that reservations must not be used to bypass waiting jobs.

4. Simulation of the Reservation Algorithm

The presented methods were evaluated through numerous simulations varying several parameters. We used the first 2000 sound batch-jobs of the *SDSC Blue Horizon* workload trace (ca. 2 weeks) [6]. Because no reservation workload was available, we transformed 200 of the normal jobs to reservations. Therefore we split the 2000 jobs into 200 equally sized subgroups maintaining the original ordering and selected randomly one job from each group to be a reservation request. This selection was done only once and all simulations were run with the same set of reservations. The duration of the reservation is derived from the actual execution time j_{et} of the job in the original workload. The studied parameters are

book-ahead time: Defines the earliest start time est of the reservation with respect to the submission time of the job to be transformed. We used 0, 2, 4, 6, 12 and 24 hours as book-ahead time in different runs. The reference value for the book-ahead time is the submission time from the original workload.

time range: Defines the flexibility in choosing a start time. We calculated the latest end time let as $est + j_{et} + k \cdot hours$, $k \in \{0, 1, 2, 5, 10, 30\}$.

processor range: Defines the flexibility in choosing the number of processors, i.e. the range of processors $[np_{min}, np_{max}]$. We used the factor pairs $\langle 1, 1 \rangle$ and $\langle 0.5, 2 \rangle$ for calculating the range relative to the number of processors of the job in the

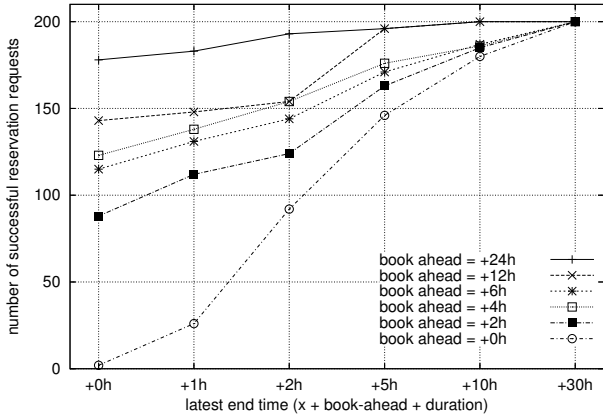


Figure 6. Number of successful reservations with respect to different book-ahead times (different curves) and time range flexibility.

original workload. If a job used 8 processors, we derived the ranges [8, 8] and [4, 16] for different simulation runs.

esr method: Defines which method is used to calculate the esr value for a reservation candidate. We used the methods *static* (with 18000 as parameter a reservation candidate, so approx. 9h30 hours book-ahead time correspond to a value of 0.9), *history* (with calculating the average usage on a daily basis) and *load* (with accuracy values of 0.5).

However, we employed a couple of restrictions to the simulations. First, only one site has been simulated. Second, only one client requested a reservation at any time (no concurrent requests). Third, we used the same user preferences for all reservation requests. The prioritized user preferences were set to 1st *end time*, 2nd *cost* and 3rd *-esr*. Hence, the earlier candidates are preferred, and so on.

Altogether, more than 400 simulations have been performed. Here we present a selection of the results, concerning mostly the efficiency of the algorithm and its impact on user jobs (i.e. the remaining 1800 jobs).

4.1. Number of Successful Reservations

Intuitively, one would expect that the larger the book-ahead time is, the larger is the number of successful reservation requests. Also, with larger time ranges the more reservation requests should succeed.

Fig. 6 shows the number of successful reservation requests with respect to the book-ahead time (different curves) and the amount of flexibility for choosing the start time (x-axis). The other parameters were load (esr method), 0.5 (walltime accuracy), $\langle 1, 1 \rangle$ (proces-

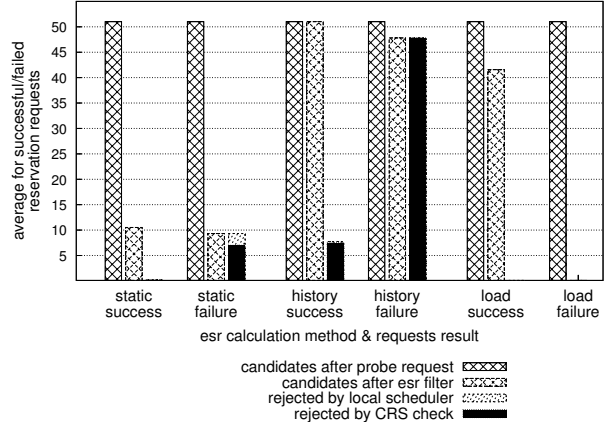


Figure 7. Performance results for reservation requests.

sor range factors). The curves confirm the expectations described in the beginning of this Section.

4.2. Efficiency of the Reservation Algorithm

The reservation algorithm should be efficient for both successful and unsuccessful reservations. Basically, the employed mechanisms follow the idea of avoiding requests that might fail. The system utilization is expressed with the esr value. After receiving the reservation candidates as response to a probe request they are processed in three steps. First, their esr value is checked against some threshold (we used 0.85). The lower the threshold is, the less candidates are filtered out. This means for the *static* method that a reservation with a book-ahead time of more than 9h30 is accepted. For the *history* method it means that a reservation is accepted if about $1.74 \cdot np$ processors (np is the number of requested processors) are available during the requested time-frame. Second, the CRS checks if the current system utilization permits the request to proceed. At this point we enforce the rule, that reservations may not overtake waiting jobs. Third, the request is passed to the local scheduler. The better the threshold filter and the CRS checks are, the less requests will fail at the local scheduler.

While *static* is trivial to implement, we expect it to yield the lowest efficiency. The method *load* should be better than *history*, because it uses the actual load situation. The detailed results (Fig. 7) are presented for simulations with 2 hours book-ahead time and 10 hours time range.

For each method two groups of three boxes are plotted. One group (left) shows the average values for successful requests. The other group (right) shows the average values for failed requests. In each group the first box (left most) represents the number of candidates received from the probe request. The second (middle) shows the number of remaining candidates after applying the threshold on the esr value.

The third box is split in two, one (the lower) for the number of reservation tries rejected by the CRS and one (the upper) for the number of tries rejected by the local scheduler. Note: Some boxes are not visible, because the corresponding value is zero. Out of the 200 elastic reservation requests, 181 were successful with the *static*, 184 with the *history* and 185 with the *load* method.

The methods *static* and *load* perform equally well for the successful requests. Both methods do only require one reservation try (no third box means no rejections by CRS and local scheduler). With the method *history* none candidate is filtered out by the threshold (mainly due to the small uncertain recorded history). Hence, quite a few (approx. 15%) are rejected by the CRS, because their book-ahead time is too small.

For unsuccessful requests the method *load* performs best. Here, all candidates are already filtered out by the threshold. Second best performs the method *static*. About 20% of the candidates remain after the threshold filtering. All these are tried out by sending reserve requests to the CRS, which rejects 3/4 of the received requests. While the method *history* filters out less candidates in the threshold step, the remaining candidates are all rejected by the CRS.

Summarizing, the method *load* performs best for all requests. The method *history* performs not as good as expected. Although, the method *static* performed quite well, one should note that this may be due to the equal book-ahead time and the relatively large time range (10 hours) of all reservation requests and the threshold at approx. 9h30.

4.3. Impact on User Jobs

Reservations reduce the scheduling flexibility for normal jobs. Before a reservation becomes active, all user jobs which access the reserved resources must have finished. Therefore the scheduler starts only jobs which are guaranteed to terminate before the reservation begins. This leads to fragmentation and user jobs experience a higher delay before they can start. According to [3], response time is a central metric for measuring job management performance. Since the execution time of the jobs is the same for every simulation run, we only used the waiting time as a metric.

The following Table lists the results for six simulation runs (with common parameters: time range 30 hours, wall time accuracy 0.5, esr method *load*, processor range factors $\langle 1, 1 \rangle$ and different book-ahead times):

Book-Ahead	0h	2h	4h	6h	12h	24h
Wait in <i>s</i>	1200	1183	1906	1871	2042	4130

All reservation requests were successful. Without any reservations the average waiting time of the batch jobs is

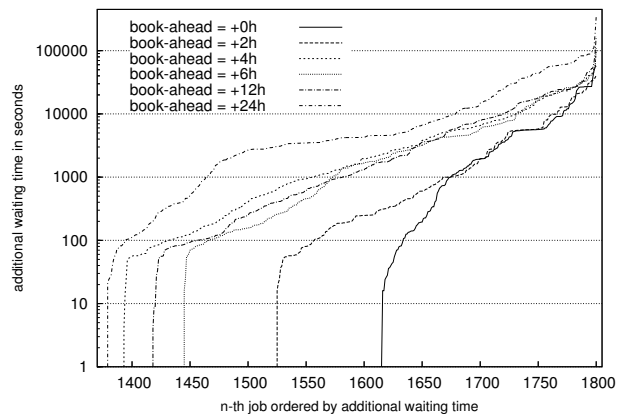


Figure 8. Additional waiting time of batch jobs for simulation runs with different book-ahead times.

approx. 700 seconds.

Even though, the number of successful reservation requests is equal for all simulations shown, the average waiting time differs significantly. A detailed analysis unveils, that most jobs experience no additional delay (this observation was also made in [11] with respect to small book-ahead times). While a few jobs wait less than in the run without any reservations, more jobs wait longer. Fig. 8 shows the jobs waiting longer (approx. 200 to 400 jobs). Each curve shows the results for a specific simulation run (i.e. book-ahead time). For each simulation run the additional waiting time was calculated for each job. Then the values were sorted in increasing order. Hence, the number on the x-axis does not refer to the actual job number in the simulation, but to its position in the additional waiting time sorted list. The graphs show that the higher the book-ahead time is, the more jobs experience larger delays.

5. Conclusion

We presented an efficient algorithm for processing elastic reservation requests in Grid environments. The algorithm supports elasticity in the duration, the start time, the number of CPUs and the site to be chosen. By introducing the 'estimated success rate' as property to eligible reservation candidates and filtering out those candidates whose esr is lower than a configurable threshold, the algorithm achieves good performance both in case of success and failure of the request.

Additionally, we studied the influence of different reservation-schema to the average waiting-time of normal batch jobs and found that the longer the book-ahead time is, the larger is the negative influence on the waiting time for other jobs.

Acknowledgements

We thank the two Junior Research Groups of the Berlin Center for Genome Based Bioinformatics at ZIB who let us run the simulations on their cluster. We thank T. Gargya from IBM Germany, Böblingen, for fruitful discussions in the early phase of this project and thank our student staff member J. Meltzer for his support during the implementation.

References

- [1] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Conference, Atlantic City, N.J., USA*, pages 483–485. AFIPS Press, April 1967.
- [2] A. B. Downey. A model for speedup of parallel programs. Technical Report, U.C. Berkeley, 1997.
- [3] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. *Lecture Notes in Computer Science*, 1459:1–24, 1998.
- [4] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.
- [5] A. G. Greenberg, R. Srikant, and W. Whitt. Resource sharing for book-ahead and instantaneous-request calls. *IEEE/ACM Transactions on Networking*, 7(1):10–22, 1999.
- [6] Parallel Workloads Archive. <http://www.cs.huji.ac.il/labs/parallel/workload/>, July 2004.
- [7] T. Roelblitz, F. Schintke, and A. Reinefeld. From Clusters to the Fabric: The Job Management Perspective. In *Proceedings of the IEEE Intl. Conference on Cluster Computing (Cluster'03), Hong Kong, China*, December 2003.
- [8] L. Smarr and C. E. Catlett. Metacomputing. *Commun. ACM*, 35(6):44–52, 1992.
- [9] W. Smith, I. Foster, and V. Taylor. Scheduling with Advanced Reservations. In *Proceedings of the IPDPS Conference, Cancun, Mexico*, May 2000.
- [10] W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 202–219. Springer Verlag, 1999.
- [11] Q. Snell, M. Clement, D. Jackson, and C. Gregory. The performance impact of advance reservation meta-scheduling. *Lecture Notes in Computer Science*, 1911:137–153, 2000.
- [12] L. Yuan, C.-K. Tham, and A. L. Ananda. A probing approach for effective distributed resource reservation. In *Proceedings of the Second International Workshop on Quality of Service in Multiservice IP Networks*, pages 672–688. Springer-Verlag, 2003.
- [13] L. Zhang, S. Deering, and D. Estrin. RSVP: A new resource ReSerVation protocol. *IEEE Network*, 7(5), September 1993.