Resource Co-Allocation for Large-Scale Distributed Environments

Claris Castillo
IBM Research
Hawthorne, NY, US
claris@us.ibm.com

George N. Rouskas
NC State University
Raleigh, NC, US
rouskas@csc.ncsu.edu

Khaled Harfoush
NC State University
Raleigh, NC, US
harfoush@csc.ncsu.edu

ABSTRACT

Advances in the development of large scale distributed computing systems such as Grids and Computing Clouds have intensified the need for developing scheduling algorithms capable of allocating multiple resources simultaneously. In principle, the required resources may be allocated by sequentially scheduling each resource individually. However, such a solution can be computationally expensive, hence inappropriate for time-sensitive applications, and may lead to deadlocks. In this work we present an efficient online algorithm for co-allocating resources that also provides support for advance reservations. The algorithm utilizes data structures specifically designed to organize the temporal availability of resources, and implements co-allocation through efficient range searches that identify all available resources simultaneously. We use simulations driven by real workloads to show that the co-allocation algorithm scales to systems with large numbers of users and resources, and we perform an in-depth comparative analysis against existing batch scheduling mechanisms. Our findings indicate that the online scheduling algorithms may achieve higher utilization while providing smaller delays and better QoS guarantees without adding much complexity.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design, Performance

Keywords

resource co-allocation, scheduling, advance reservation

1. INTRODUCTION

The development of large scale distributed computing systems such as Grids and Computing Clouds has experienced enormous growth in recent years. Furthermore, technological advances in virtualization and new programming paradigms such as MapReduce [11] have catalyzed the adoption of such systems as the *de facto* infrastructure for computing service provisioning in academia and corporate R&D environments. At the same time, we are witnessing the emergence of a wide range of applications that capitalize on the availability of distributed computing to perform complex tasks with strong temporal and spatial requirements. For instance, several scientific workflow applications [23, 13, 26] involve the orchestration of multiple computation and data transfer stages. These stages normally have strong dependency on completion times; thus the ability to co-schedule and synchronize resource usage becomes crucial. Within this group, an emerging class of deadline-driven scientific applications such as severe weather modeling [31] require simultaneous access to multiple resources and predictable completion times.

More recently, MapReduce [11] has been introduced and widely adopted within the Cloud Computing community as a programming framework for massively parallel jobs. This new paradigm harnesses the aggregation of computing power across networks to process large amount of unstructured data in time scales never envisioned before. To this end, the MapReduce middleware (e.g., Hadoop [28]) allocates multiple compute nodes to run multiple instances of a set of functions defined by the user. Consequently, the ability to allocate efficiently multiple resources from a large pool is crucial for this paradigm to achieve its full potential. The importance of such applications has been recognized by research, industry and government communities through the support of several initiatives, including several that focus on developing infrastructures that enable sharing of resources across research and academic institutions [29, 14, 4], as well as an industry-academia partnership [1] aiming to prepare the next generation of computer scientists to program at an Internet-scale using the MapReduce paradigm.

The design and development of scheduling techniques for co-allocating multiple resources plays a crucial role in the full realization of these initiatives and it is the main focus of this work. Much of the work done in co-allocation of resources in the Grid community has focused on the administrative aspects resulting from having resources distributed across multiple sites (domains). Under this assumption additional infrastructure and architectural solutions are crucial [36]. In our work we aim at developing generic scheduling algorithms that can be applied to distributed environments with different characteristics. Note that in principle, the simultaneous allocation of computing resources can be achieved sequentially, i.e, considering the request for each resource as an atomic transaction. Nevertheless, such a solution can be computational expensive and incurs in delays, and hence inappropriate for time sensitive applications [10].

HPDC'09, June 11–13, 2009, Munich, Germany.

In one of the preliminary works in this area [10] authors presented DUROC, a generic resource management framework to support co-allocation of resources in multi-site Grids. The main focus of this work was to develop mechanisms to handle the monitoring, control and configuration of distributed resources. The mechanisms presented as part of this framework were investigated by means of simulation in [5] under the assumptions that jobs have a (starting) deadline. Although we do not focus on providing deadline support for co-allocation of resources, our work can be easily extended to support deadlines as explained later in Section 5.2. In [36] a set of adaptive selection strategies are introduced to co-allocate resources. In this study authors utilize a batch scheduler with backfilling support to schedule incoming jobs. As explained later in this section, batch scheduling techniques typically implement first-come-first-serve (FCFS) policies and are difficult to extend to support advance reservations. Pure FCFS policies lead to high fragmentation of resource, low utilization of resources and limited scheduling flexibility [17]. An heuristic based on the well known list scheduling heuristic HEFT has been proposed in [12] to co-allocate resources for Grid workflows. Since HEFT is a list scheduling technique each resource request is treated atomically and hence its running time is linear to the number of slots available in the system. In our work we consider large-scale distributed systems; therefore, scalability is a key parameter in the design of our algorithm.

One problem that shares much similarities with the co-allocation problem in Grids is scheduling in parallel computing [18, 19] since it also involves the allocation of multiple processors to execute a given job. As a consequence the problem of co-allocating computing nodes in large scale distributed environments is often tackled using techniques similar to the ones applied to parallel job scheduling–relying on batch schedulers to scheduling incoming jobs–. In fact, most resource managers in existing Grid infrastructures interact with batch schedulers such as LSF [30] and Maui [7] (via Torque [27]). Furthermore, Hadoop-on-demand [28] has also followed this approach by adopting Torque [27] and its batch scheduler Maui [7] to allocate computing nodes for MapReduce jobs. The framework has also been extended to use the batch scheduler Condor [2]. Much work has been done in parallel computing [18, 19]. Existing parallel scheduling mechanisms can be classified as *resource-driven* or *job-driven*. Batch schedulers [25, 34, 35, 24] fall into the resource-driven category since incoming jobs are queued as they arrive in a first-come, first-serve fashion and considered for scheduling whenever resources become available. With additional functionality (e.g., backfilling, priority queues) the scheduler can leverage knowledge about jobs currently executing (e.g., their completion times) to make scheduling decisions that optimize for system utilization or user's priority. Similarly, in [17] the author investigated the performance impact of multiple packing schemes in gang scheduling. Slots are considered serially, however and hence, the duration of the algorithm can be significantly long in large distributed environments. Online schedulers [22, 21], on the other hand, fall into the job-driven category as new jobs are scheduled to resources as soon as they arrive into the system. The algorithm we present in this paper attempts to combine the best of both worlds: it schedules incoming jobs as soon as they arrive (providing users with time guarantees and finer control of their applications) and keeps a look-ahead until the horizon of the schedule (allowing for better scheduling decisions).

We also believe that advance reservations, i.e., the ability to allocate resources in advance, is one mechanism that can be used to address the lack of QoS support in batch schedulers; furthermore, such a feature also enables support for workflow applications – key applications for automation of computing distributed infrastructure. Nevertheless, designing scheduling algorithms that support advance reservations in Grid environments has proven difficult, with scalability being the main challenge faced in their design. More specifically, as systems increase in size and complexity it becomes harder to organize and maintain efficiently the large number of reservations. We refer the reader to [8, 32, 9, 20, 15] for some of the most recent and relevant work in this field.

In this work we present an online co-allocation algorithm that is effective in co-allocating resources while providing support for advance reservations and temporal range search; the latter feature allows users to easily retrieve all the resources available within a specified time window. The novelty of our approach lies in organizing the resource availability in a data structure consisting of specially designed 2-dimensional trees, in a manner that allows a single search operation to identify all required resources efficiently. The rest of the paper is organized as follows. In Section 2 we describe the online scheduling problem we study in this work. A set of applications suitable for the problem in consideration are presented in Section 3. In Section 4 we provide additional details on the implementation of the scheduling algorithm and of the data structure related to managing the temporal availability of resources. In Section 5 we present simulation results to evaluate the various strategies in terms of several performance metrics, and we conclude the paper in Section 6.

2. PROBLEM DESCRIPTION

Consider a scheduler $\mathcal{S}$ for a computing system with N servers that may be geographically distributed in a network. A user with a job requiring service submits a request r to $\mathcal{S}$. The request is characterized by a four-parameter tuple (q_r, s_r, l_r, n_r), where:

- q_r is the *request time*, i.e., the time the request is submitted by the user;
- $s_r \geq q_r$ is the *earliest time* the job can start execution, with $s_r > q_r$ permitted *if and only if* the system supports advance reservations;
- l_r is the *temporal size* of the reservation, i.e., the estimated duration of the job; and
- n_r is the *spatial size* of the reservation, i.e., the number of servers required for the given job.

We assume that $\mathcal{S}$ maintains a schedule that records, for each of the N servers, the time periods in the future during which the server is reserved for requests that have already been accepted by the system. In essence, this schedule represents the set of commitments that the system has made, and in the absence of preemptive actions (as we assume here), it guarantees that server resources will be available to the accepted jobs at specific future times. We let H denote the *time horizon* of the system, denoting how far in the future the system may schedule resources. The value of H depends on the type of applications that the system supports, and may be in the order of weeks or months. Note that the *a priori* knowledge of the temporal size of a job is a common practice in most current Grid like environments. The accuracy of this parameter has been widely investigated in the past and is out of the scope of this paper.

Figure 1 shows an example schedule for a 4-server system with a horizon of $H = 42$ time units. The schedule shows that at the current time (i.e., time $t = 0$ in the figure), there are two jobs scheduled for server 1: job A which is currently in service and will end at time $t = 4$ and job B which has

reserved the server from time $t = 25$ to $t = 34$; similarly, two jobs have been scheduled for server 2, server 3 and server 4, respectively.

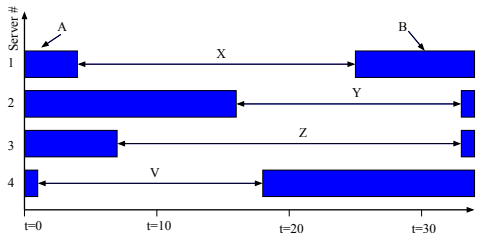

Figure 1: Sample schedule for a 4-server system with co-allocations and advance reservations

When a service request $r = (q_r, s_r, l_r, n_r)$ for a new job arrives, $\mathcal{S}$ immediately runs an algorithm to determine whether it is feasible to schedule the job starting at time $t = s_r$. If so, $\mathcal{S}$ selects a set of n_r servers that can handle this job, updates its schedule, and returns a reference to the n_r servers to the user. If, on the other hand, sufficient resources are not available starting at time $t = s_r$, the system makes an attempt to reserve resources for this request at a time $t > s_r$. To this end, the scheduler runs the same algorithm to determine whether it is feasible to schedule the job starting at time $t = s_r + \Delta_t$, where Δ_t is a configurable parameter. The scheduler repeats this process by incrementing the previous starting time by Δ_t time units until either a feasible starting time is found, or a maximum number R_{max} of scheduling attempts has been reached, whichever occurs first. The quantity R_{max} is also a configurable parameter, and is such that $R_{max}\Delta_t$ is an upper bound on the delay that applications may tolerate.

The scheduling decision influences the performance perceived by users as reflected by (1) the fraction of jobs accepted by the system, and (2) the turnaround time of the jobs (which includes any delays in their starting time introduced by the operation of the scheduler described above). It also impacts the overall system utilization, which is a measure of how well the overall capacity of the system is being used. The challenge, therefore, is to develop efficient co-allocation algorithms that minimize the fraction of delayed jobs, maximize utilization, and allocate resources fairly among users.

3. APPLICATIONS

Resource co-allocation is of interest to many applications. In this section, we discuss two applications within the context of distributed computing systems. These applications are being introduced in real working environments but their wider deployment and ultimate success depends critically on the existence of efficient co-allocation mechanisms.

3.1 The Virtual Computing Laboratory (VCL)

Computing over the Internet is becoming increasingly popular. Due to emerging infrastructures such as Grids and Computing Clouds, it is possible to develop applications that support various Internet-wide collaborations by seamlessly harnessing appropriate resources. The virtual computing laboratory (VCL) [4], an initiative established at North Carolina State University, aims to provide all K-20 institutions across the state of North Carolina with access to a wide array of distributed, high performance computing and networking resources, in support of research and education. Given the large number of resources and users to be accommodated, scalable management and provisioning of resources has emerged as a major challenge for this project.

The VCL infrastructure is designed to provide a suite of distributed applications to the state's educational institutions. Two applications directly related to our work include desktop virtualization and high performance computing (HPC), both of which require efficient tools for automated reservation and scheduling of resources. Desktop virtualization consists of dispatching virtual desktops, customized to a set of specific requirements, for in-class and laboratory instruction. This initiative provides schools with access to cutting-edge technology that can be leveraged to support advance educational techniques (e.g., learn by playing, virtual environments) and that would otherwise be impossible for individual schools to afford or deploy. The second major application of VCL, namely the provisioning of HPC resources for large-scale jobs, targets more sophisticated and demanding users such as graduate students, scientists and faculty who rely on computationally intensive experiments to carry out their research.

In both applications, users submit requests to obtain exclusive use of multiple resources over a specific time window based on class schedules and/or job-specific deadlines. The resource manager then runs an algorithm to determine the availability of the resources and informs the user. If the request is granted, the manager sends the authentication information required for the user to gain access to the resources. Otherwise, it suggests alternative times at which the resources are available. The algorithm we develop in this work supports both on-demand requests (appropriate for workloads with best-effort requirements) and advance reservations requests (appropriate for guaranteeing the availability of resources at specific future times, e.g., during class hours). Therefore, it is suitable for the VCL and similar environments that are characterized by mixed workloads.

3.2 Lambda Scheduling for Grid Applications

Current research practices tend towards collaboration among institutions across countries that require high bandwidth connections to use multiple network-connected resources. This need has been recognized by the government and several initiatives have been funded and established to promote such collaboration [14]. The realization of this vision depends on (1) the deployment of infrastructure that provides automated and rapid establishment of lambdas across administrative domains, and (2) the development of scheduling mechanisms that allow researchers to request and manage network resources on demand.

Critical to this vision is the scheduling of link wavelengths within each administrative domain along an end-to-end network path. One proposal that has gained popularity is the deployment of a path computation element (PCE) [16, 6] as a (semi-)centralized entity to handle the scheduling of link wavelength resources. This scheduling problem can be informally expressed as follows: *Given a request consisting of a source-destination node pair, a range of wavelengths, a time window, and the estimated length of the connection, find a path and associated wavelength (or wavelengths, if wavelength conversion is available) from the source to the destination nodes to satisfy the request.*

Since the wavelengths(s) on all links of the path must be allocated and de-allocated *simultaneously*, this problem falls in the class of resource co-allocation problems.

4. RESOURCE CO-ALLOCATION ALGORITHM

The algorithm we describe in this section has three important features of value to applications similar to the ones we described above. First, its low complexity enables the resource manager to provide short response times, which in turn improves the overall efficiency of the system. Second, the operation of the algorithm is based on performing a range search for identifying the available resources. The range search returns all the resources available within the specified time window. Third, it supports advance reservations and can be easily extended to support deadlines. As a result, users may use sophisticated post-processing techniques to optimize the selection of resources based on their requirements or other criteria. For instance, a Grid application may run customized routing algorithms to select among the available paths and wavelengths based on cost, data rate, or other considerations.

In the rest of this section, we first describe the data structure used to maintain and update the availability of resources, and we then describe the range search algorithm and analyze its complexity.

4.1 Data Structure for Temporal Resource Availability

We define an *idle period* as a time period during which a server is idle and hence available for service. Our objective is to organize the idle periods in a way that enables efficient search and update operations. To this end, we partition the temporal space into a set of Q slots of size τ, where $Q = \lceil \frac{H}{\tau} \rceil$ and H is the time horizon of the system. Quantity τ is taken as the unit of time. Without loss of generality, we assume that τ is equal to the minimum temporal size of reservation requests. (Note that jobs of size smaller than τ may be packed together and submitted through a single request of size at least equal to τ.) The significance of imposing a minimum temporal size is discussed shortly.

The temporal availability of servers (i.e., the collection of the corresponding idle times) is organized in a data structure that consists of Q 2-dimensional trees $T_q, q = 1, \ldots, Q$, one for each slot within the time horizon H. We let T_q^s and T_q^e denote the tree corresponding to the first and second dimension, respectively, of tree T_q. Tree T_q^s stores in its leaf nodes all the idle periods that span slot q in *descending order of their starting time*. Specifically, the leaf node corresponding to idle period i records the following information for i:

- its starting time st_i;
- its ending time et_i; and
- an identifier id_i that is used to identify the server on which this idle period occurs.

On the other hand, each internal node u of tree T_q^s stores the following information:

- the median starting time of the idle periods stored in the subtree of T_q^s rooted at u;
- the size of the subtree rooted at u; and
- a pointer to a secondary binary search tree $T_q^e(u)$.

Trees $T_q^e(u)$, where u is an internal node of tree T_q^s, correspond to the second dimension of the 2-dimensional tree structure. The leaf nodes of $T_q^e(u)$ store the idle periods in u's subtree of the primary tree T_q^s, in *ascending order of their ending time*. Each internal node v of tree $T_q^e(u)$ records the following information:

- the median ending time of the idle periods stored in the subtree $t_q^e(u)$ rooted at v; and
- the size of the subtree rooted at v.

Figure 2 provides an example to illustrate these concepts. Figure 2(a) shows the idle periods, denoted by X, Y, Z, and V, of the 4-server system shown in Figure 1, and a slot size $\tau = 10$. For instance, idle period X has starting time $st_X = 4$, ending time $et_X = 25$, and corresponds to server $id_X = 1$. Figure 2(b) shows the 2-dimensional tree for slot $q = 2$ that corresponds to the time interval [10,20]. Since all four idle periods overlap (at least partially) with this slot, the primary tree T_2^s stores all four in its leaves in decreasing order of their starting times. Since the median of the starting times of these idle periods is 7, this is the value stored in the root A of the tree. The secondary tree $T_2^e(A)$ corresponding to the root A of the primary tree also stores all four idle periods in its leaf nodes, but in increasing order of their ending times. The median (=25) of the four ending times is stored in the root of this tree. Similarly for the secondary tree $T_2^e(B)$ corresponding to node B of the primary tree (the secondary tree $\mathrm{T}_2^e(C)$ corresponding to node C of the primary tree is not shown in the figure).

Note that in this data structure, an idle period is stored in the trees of *all* slots with which it overlaps; for instance, idle period V in Figure 2 appears in the trees for slots $q = 1$ and $q = 2$. Furthermore, if the slot length τ is equal to the minimum spatial job size, then the number of idle periods stored in each tree is bounded to the number of servers in the system N, i.e., each tree will include at most one idle period per server.

Finally, we note that as the time advances, the tree corresponding to the just expired time slot is discarded, and a new tree is created (initialized) for the new slot at the end of the system's time horizon; as a result, the system always maintains Q trees, with each tree containing at most N idle periods. These discard and initialization operations are repeated every τ time units and take $O(1)$ time.

4.2 Online Scheduling Algorithm

We now present the online scheduling algorithm by describing how it handles a reservation request $r = (q_r, s_r, l_r, n_r)$, where the four parameters were defined in Section 2; we also denote the ending time of r as $e_r = s_r + l_r$. Given this request, the algorithm needs to find n_r feasible idle periods for the corresponding job. An idle period $i = (st_i, et_i)$ is *feasible* for request r if and only if $st_i \leq s_r$ and $et_i \geq e_r$. On the other hand, we will refer to an idle period i that meets only the first condition of feasibility (i.e., $st_i \leq s_r$) as a *candidate* idle period.

Let q be the slot within which the starting time s_r of the request lies, and T_q^s the primary tree containing all the idle periods overlapping with this slot. To find n_r feasible idle periods, the algorithm proceeds in two phases as follows.

Phase 1. The algorithm first searches tree T_q^s to locate every candidate idle periods in slot q. To this end, the algorithm starts at the root of the tree and follows the left or right subtree based on the value of s_r. Specifically, if the median starting time at the current node is greater than s_r, the algorithm ignores the left subtree and continues to search recursively the right subtree. This is because all the idle periods contained in the left subtree have starting times that are larger than s_r, hence they are not candidates for this request. If, on the other hand, the median starting time at the node is smaller than s_r, the algorithm marks the right subtree and continues to search in the left subtree in a recursive fashion. Note that, in this case, all the idle periods

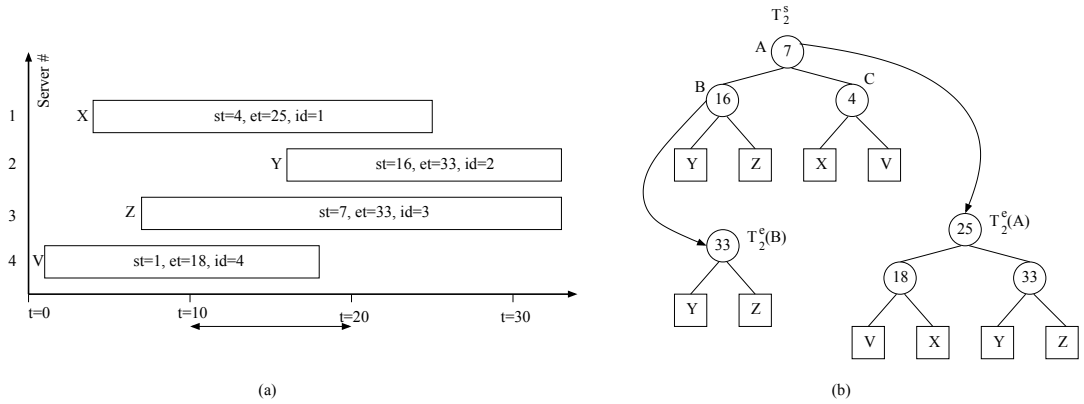

Figure 2: (a) Idle periods X, Y, Z and V in a 4-server system, and (b) 2-dimensional tree containing the idle periods in slot $q = 2$ corresponding to the time interval [10,20]; the secondary tree corresponding to node C is not shown

contained in the right subtree start earlier than s_r, therefore they are all candidates. The algorithm stops when it reaches a leaf node; if the idle period stored in this leaf is a candidate, it is also marked.

At each step of the tree search, the algorithm keeps track of the number of candidate idle periods by adding up the sizes of all marked subtrees (recall that each internal tree node stores the size of its subtree). If at the end of this phase the algorithm has found at least n_r candidate idle periods, it proceeds to the next phase. Otherwise, it can be concluded that no sufficient resources exist for the job to start at time s_r. Rather than rejecting the request, our approach in this case is to repeat Phase 1 for a modified request with starting time $s_r + \Delta_t$ in an attempt to schedule the job after a delay of Δ_t units. The value of Δ_t may be tuned by the administrator so as to optimize system and application performance. For instance, applications with tight delay requirements may request the scheduler to be aggressive in scheduling their workloads, i.e., use small values of Δ_t. Nevertheless, the algorithm may take longer to find a schedule. The algorithm repeats Phase 1 a maximum of R_{max} times, each time increasing the starting time of the request by Δ_t.

Let us return to the example presented in Figure 2, and consider a request $r = (q_r = 17, s_r = 17, l_r = 12, n_r = 2)$. The algorithm searches the tree T_2^s in Figure 2(b) starting at the root A. Since the median starting time value stored in A (=7) is smaller than s_r, the right subtree containing the candidate idle periods X and V is marked. The algorithm continues its search in the left subtree rooted at node B, marks the tree containing candidate idle period Z, and stops when it reaches the leaf containing idle period Y (which is also a candidate idle period and hence is marked as well). Since the algorithm has found $4 > n_r = 2$ candidate idle periods, it proceeds to Phase 2.

Phase 2. Having found a set of at least n_r candidate idle periods in Phase 1, the algorithm next searches for those idle periods i in the set that also meet the condition $et_i \geq e_r$ and are thus feasible for the given request r. To do this, the algorithm searches the secondary trees $T_q^e(u)$ associated with the nodes (subtrees) u of the primary tree T_q^s marked in Phase 1; these subtrees are searched in *reverse order* in which they were marked.

The search algorithm for tree $T_q^e(u)$ proceeds as follows. Starting at the root, if the median ending time stored there is larger than e_r then the right subtree is marked and the search continues recursively in the left subtree; in this case, all candidate idle periods in the right subtree are also feasible. If the median ending time is smaller than e_r, the algorithm ignores the left subtree (since all idle periods contained therein are not feasible) and proceeds to search recursively in the right subtree. The algorithm stops whenever it reaches a leaf node or whenever it has identified at least n_r feasible idle periods, whichever happens first. If n_r feasible idle periods are found, an in-order traversal of the subtrees marked in this phase is invoked to retrieve the feasible idle periods. Otherwise, the search fails; in this case, if the maximum number R_{max} of scheduling attempts has not been reached, the algorithm continues from Phase after incrementing the starting time of the request by Δ_t time units.

Returning to the example in Figure 2 with request $r = (q_r = 17, s_r = 17, l_r = 12, n_r = 2)$, recall that the search of the primary tree marked the subtrees rooted at A, Z, and Y, in this order (the latter two nodes are leaves). Hence, in Phase 2, the algorithm searches node Y first, and confirms that it is a feasible idle period; it then repeats the process with node Z, which also corresponds to a feasible idle period. At this point, the algorithm has found $n_r = 2$ feasible periods and returns them.

For each feasible idle period allocated to a new job, the algorithm needs to update the data structure so as to: (1) remove the idle period from the trees of all slots it overlaps; and (2) add any new idle period created. Note that if an idle period $i = (st_i, et_i)$ is allocated to a request r with starting and ending times s_r and e_r, respectively, at most two new idle periods will be created: $j = (st_i, s_r)$ and $k = (e_r, et_i)$. Inserting or removing an idle period takes time that is logarithmic in the size of the corresponding tree, and this size is bounded by the number N of servers. Note also that this update process may be implemented in the background to minimize its impact on the performance of the scheduler.

Range Searches. As we mentioned earlier, this algorithm makes it possible to perform a range search for resources. This is a desirable feature as it permits users to strategically select resources that optimize their applications and/or meet specific preferences. For example, a user that is interested in reserving resources within a time window $[t_a, t_b]$ may submit a request such that $s_r = t_a$, $l_r = (t_b - t_a)$ and $n_r \geq 1$. The scheduler runs a simplified version of the algorithm and returns the set of resources available (if any) in this window, without updating the tree data structures. The user may then run an application-specific algorithm to select a subset

of these resources that optimize some aspect of the application, and contact the scheduler to commit the resources resulting from this selection. Due to space constraints we do not investigate this feature of the scheduling algorithm in this paper.

4.3 Algorithm Complexity

Each invocation of Phase 1 of the algorithm takes time at most $O(\log N)$ since the binary search tree T_q^s may contain at most N idle periods, one for each server. In this phase, the algorithm marks at most $\log N$ subtrees to be searched in Phase 2. Searching each of the $\log N$ secondary trees in Phase 2 takes time $O(\log N)$; hence, in the worst case, the search in Phase 2 takes time $O((\log N)^2)$. Assuming that the algorithm finds n_r feasible idle periods, it invokes an in-order traversal to retrieve them in time $O(n_r)$. In addition, the algorithm needs to update each of the trees containing the n_r feasible idle periods that are now allocated to the current job. This update operation takes $O(n_r \times Q \times (\log N)^2)$ time, and takes place only when the scheduling attempt is successful. Therefore, the overall complexity of the algorithm for a successful scheduling attempt is $O(n_r \times Q \times (\log N)^2)$. Note that the algorithm may make up to R_{max} unsuccessful scheduling attempts, each of which takes $O((\log N)^2)$ time in the worst case.

5. PERFORMANCE EVALUATION

We use simulation experiments to evaluate the performance of the co-allocation algorithm we described in the previous section and to compare it to batch scheduling algorithms that are typically used to allocate resources in Cloud and Grid computing systems. To this end, we use three real workloads, obtained from the parallel workload archive [3], to drive the simulations. Table 1 summarizes the pertinent features of the three workloads; these are representative of medium-size Grid systems in use today, have been fully sanitized, and have been used in numerous research studies [3].

All three systems shown in Table 1 implement some variant of a batch scheduler where jobs are placed into one or multiple queues waiting for resources to become available before execution. Each log entry in the traces corresponds to a single job and contains information about the job such as starting time, expected running time, submission time, ending time, number of processors, user id, computer id, and waiting time. Recall that in our model a request r is represented by the four-parameter tuple (q_r, s_r, l_r, n_r). Therefore, for the purpose of this study we extracted the same four parameters from each log entry. We note that events such as servers going down for maintenance are difficult to infer from the workload traces. Nevertheless, we feel that such events have little impact on the results overall due to the variety of workloads and their large size. We also set the maximum number of scheduling attempts $R_{max} = Q/2$, i.e., equal to one-half the number of slots in the time horizon, and the increment by which the starting time of a job increases in subsequent scheduling attempts to $\Delta_t = 15$ minutes. The value of Δ_t may impact the overall performance of the system and therefore should be tuned as we mentioned earlier. Its value in our experiments was set empirically, i.e., no major gain was found for smaller values of Δ_t.

In our study we use three performance metrics:

- *Waiting time,* W_r, is a measure of the QoS perceived by the user, and refers to the time between the earliest time the job can start execution ($= s_j$) and the actual time it starts execution under a given scheduler.
- *Temporal penalty,* P_r^l, is a measure of the fairness experienced by the user and is defined as: $P_r^l = \frac{W_r}{l_r}$. In other words, P_r^l is the value of the waiting time normalized to the duration of the job. Intuitively, lower values correspond to a more fair treatment of jobs. Ideally, values $P_r^l < 1.0$ are desirable. In practice, however, this usually does not hold for small jobs.
- *Spatial Penalty,* P_r^n, is also a measure of the fairness experienced by the user and is represented by the average W_r as a function of the spatial size ($= n_r$) of the job. Intuitively, the larger the number of resources needed by a given job, the harder it is for $\mathcal{S}$ to schedule the job and therefore, the longer the penalty in terms of waiting time.

We organize our performance evaluation in two parts. First, we investigate the performance of our online algorithm against the performance deduced from the traces where batch scheduling is used. Second, we study the impact of introducing support for advance reservations in our algorithm against batch scheduling algorithms.

5.1 Online Co-Allocation of Resources vs. Batch Scheduling

Let us first investigate the difference in performance between our online co-allocation algorithm and the batch scheduling algorithms used for the workloads shown in Table 1. For a fair comparison, we let $q_r = s_r$ for every job request so that the co-allocation algorithm attempts to schedule each job as soon as the corresponding request is received. If no sufficient resources are found starting at time $t = s_r$, the algorithm makes up to R_{max} additional scheduling attempts, each time incrementing s_r by Δ_t time units, as described in Section 4.2.

Figure 3(a) plots the temporal penalty P_r^l experienced by jobs of the KTH workload under the batch and online scheduling algorithms. We observe that small jobs experience a higher temporal penalty, an order of magnitude or more, under the batch scheduler compared to our online co-allocation algorithm. A more careful look at the mid-tail of both curves (for jobs of size between 2 to 10 hours) in Figure 3 (b), on the other hand, reveals that our algorithm penalizes larger jobs more heavily. Similar results were obtained for the other two workloads, and are omitted due to space constraints. These results are somewhat counter-intuitive considering that most batch schedulers implement some sort of backfilling, i.e., allow small jobs to leap ahead in the queue as long as they don't delay the job at the head of the queue, and therefore it is expected that small jobs would experience a relative lower degree of penalty. A more detailed analysis of the traces reveals that our algorithm is more efficient in finding idle periods to allocate incoming small jobs without delaying them much. This is reflected by the small number of times that the algorithm needs to reschedule (by increasing s_r) small jobs as observed in our simulations; we further discuss this observation later in this section. On the other hand, batch schedulers find it difficult to identify resources available to fit small jobs due to the high fragmentation of resources and the dominant presence of jobs that are large in spatial and temporal dimension.

Figure 4(a) plots the waiting time distribution for both algorithms under two workloads, CTC and KTH. Let us first consider the curves for the CTC workload. Under our online scheduler most jobs have a waiting time smaller than 2 hours. This is an improvement of nearly a factor of two when compared to the batch scheduler. Furthermore, the tail lengths of the two curves differ by hundreds of hours,

Table 1: Features of workloads used in the performance evaluation.

Workload	No. of processors (N)	No. of jobs	Avg. estimated l_r (hours)
CTC	512	39,734	5.82
KTH	128	28,481	2.46
HPC2N	240	202,825	4.72

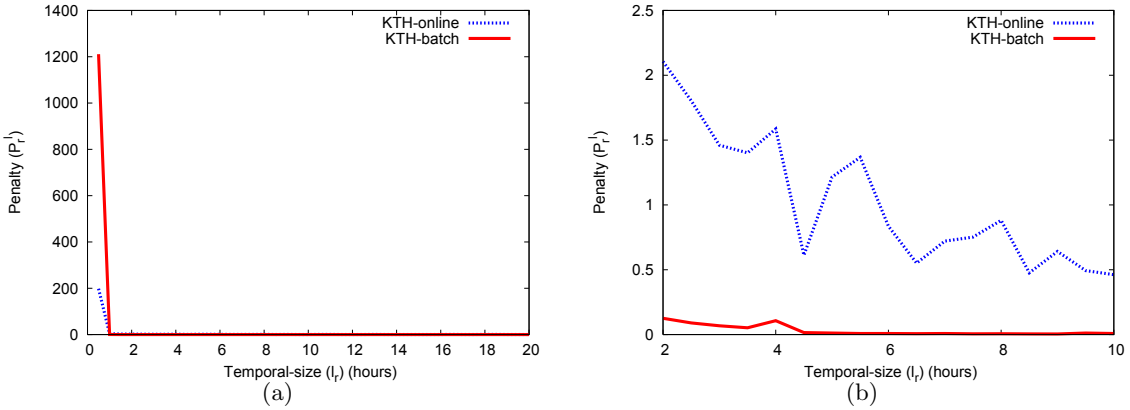

Figure 3: Temporal penalty P_r^l for the KTH workload: (a) all jobs, (b) medium-size jobs

with the maximum waiting times being 19 hours for our online scheduler and a much higher 674 hours for the batch scheduler. The results for the KTH workload are slightly different in that more jobs wait for less than 1 hour under the batch scheduler as compared to our algorithm. However, the waiting time distributions even out before 2 hours. Similar to the CTC workload, the length of the tails differ by two hundred hours, with the maximum waiting time for the online and batch schedulers being 75 hours and 272.5 hours, respectively.

We make two major observations from these results. First, as shown in Figure 4(b), most jobs in the KTH workload have a duration smaller than 2 hours. This results in higher resource fragmentation, impeding the scheduling algorithm from finding feasible idle periods to allocate for incoming jobs. This is in contrast to the CTC workload where at most 14% of all jobs are smaller than 2 hours. This suggests that the performance of our algorithm is not oblivious to the workload, which is a common observation in most scheduling algorithms. Second, the large difference in the tail length for both workloads suggests that by keeping a look-ahead view till the time horizon H, our algorithm can pack incoming jobs more efficiently and hence improve the utilization of the system.

In Figure 5 we plot the average waiting time W_r as a function of job spatial size for two workloads. We observe that the waiting time increases with the spatial size for both online and batch algorithms. However, our algorithm achieves smaller average waiting time compared to the batch algorithm for both workloads. This observation follows from the results presented in Figure 4 and can be explained similarly: our algorithm can pack jobs efficiently by taking advantage of the look-ahead view till the horizon H.

We conclude this part of our evaluation by presenting in Table 2 the number of scheduling attempts that our algorithm makes per request, as a function of the spatial size for workloads CTC and KTH. In order to obtain results that are statistically significant, we calculate the average number of scheduling attempts as a function of n_r in groups of 50 servers. For instance, the first column of the table corresponds to the average number of scheduling attempts for jobs such that $0 < n_r \leq 50$. Empty spaces in the table represent the case in which there were no requests with n_r values within the corresponding range. It can be observed from the table that, as n_r increases, the number of scheduling attempts increases as well. This can be explained from the fact that as jobs demand more resources, the fragmentation in the system increases and it becomes harder for the algorithm to schedule incoming requests. We also observe a larger number of scheduling attempts for the KTH workload as compared to CTC workload. This is due to the fact that KTH exhibits higher fragmentation as a result of the temporal size distribution observed in Figure 4(b).

5.2 Online Co-Allocation of Resources vs. Batch Scheduling with Support for Advance Reservations

Due to the fact that advance reservations are not widely implemented in existing systems, there are no workload traces in the Parallel Workload Archive [3] that represent the advance reservation model. In order to evaluate the performance of our algorithm we generated advance reservation requests by randomly selecting jobs from the workload traces according to a desired proportion of advance reservations in the experiment. We denote the fraction of jobs with advance reservations in the system by $\rho \leq 1$. For any advance reservation request we randomly set its requested start time (s_r) to be within zero to three hours in the future, as in the study presented in [33]. Note that the algorithm can be easily extended to support user's deadline by setting the starting time to the earliest time a given job needs to start to meet the deadline imposed by the user.

Figures 6(a) and (b) show the waiting time distribution for different values of ρ for workloads CTC and KTH, respectively. In both graphs we observe a peak around 3 hours: this is a consequence of setting the requested start time to be between zero and three hours as mentioned earlier. We observe that as ρ increases, the distribution of waiting times

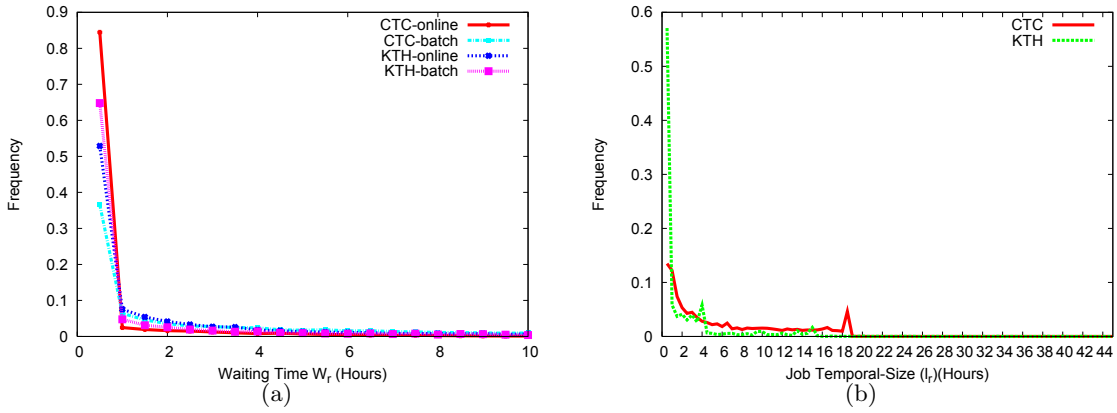

Figure 4: (a) Waiting time (W_j) distribution (CTC and KTH). (b) Temporal-size distribution (l_r) for workloads CTC and KTH.

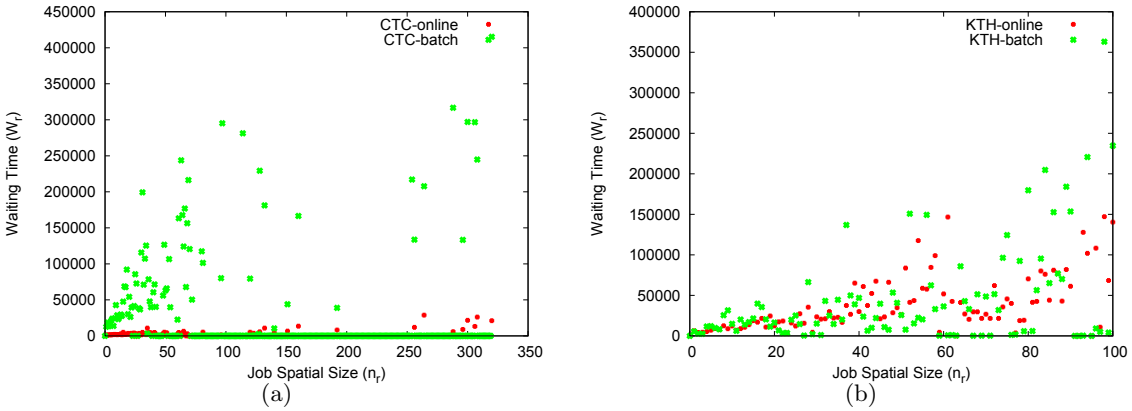

Figure 5: Average waiting time (W_r) as a function of job spatial size: (a) CTC workload, (b) KTH workload

changes in the range [0:3] hours for both CTC and KTH. However, the tail lengths for both remain the same. We also note that under the CTC workload our algorithm outperforms the batch scheduling algorithm for multiple values of ρ. This is in contrast to the results plotted in Figure 6 (b) for KTH where batch scheduling performs better than our co-allocation algorithm for all values of ρ. Nevertheless, as we mentioned in our discussion of Figure 4, the tail for the batch scheduler is significantly longer than for the online algorithm.

Figure 7(a) presents the average waiting time against $\rho, 0 \leq \rho \leq 1$, where $\rho = 0$ corresponds to the case of no advance reservations while $\rho = 1$ represents the case in which all jobs use advance reservations. We observe that the waiting time increases as ρ increases. This result is in agreement with intuition since by increasing ρ we effectively increase the waiting time of a larger fraction of jobs.

To evaluate the scalability of our algorithm in the presence of advance reservations, Figure 7(b) depicts the average number of computational operations performed by the scheduling algorithm to schedule a request r as a function of ρ, under the three workloads. The graph shows that our algorithm scales well as the fraction of advance reservations increases. The reasoning behind this observation is that when performing advance reservations, it is more likely that the algorithm will find resources available without having to search in multiple slots, resulting in fewer scheduling attempts. On the other hand, when scheduling incoming jobs immediately, i.e., $s_r = q_r$ the scheduler is more likely to search additional slots after searching in the slot containing s_r. Therefore, even though increasing ρ increases resource fragmentation in the system, and hence the size of the binary search trees is expected to increase, the number of operations remains relatively constant due to the smaller number of slots searched and the balancing feature of the binary search trees being used.

6. CONCLUSIONS

In this paper we considered the problem of co-allocation of resources in large-scale distributed systems. We have developed an online co-allocation algorithm that is effective in co-allocating resources while providing support for advance reservations and range searches. Our approach combines an efficient data structure of multiple 2-dimensional trees to organize the temporal availability of system resources with efficient searches to locate all available resources simultaneously. We also performed an in-depth comparative analysis of our algorithm against conventional batch schedulers under real workloads. Our results provide some insightful conclusions indicating that, under typical conditions, our scheduling algorithm achieves high utilization and provides smaller delays without adding much complexity. We also

Table 2: No. of scheduling attempts as a function of spatial size, CTC and KTH workloads

Workload / n_r	(0:50]	(50:100]	(100:150]	(150:200]	(250:300]	(350:400]
CTC	2.96	5.34	7.22	13.25	—	127.44
KTH	10.27	60	120	—	—	—

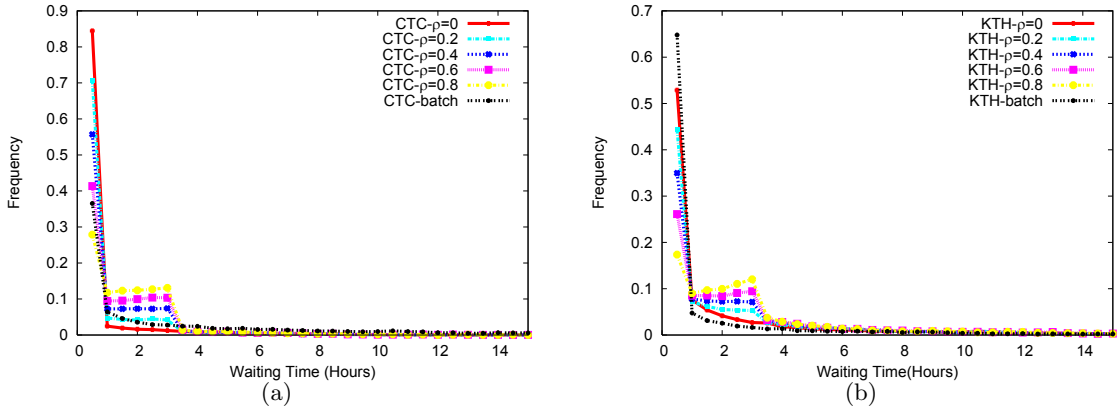

Figure 6: Waiting time distribution: (a) CTC workload, (b) KTH workload

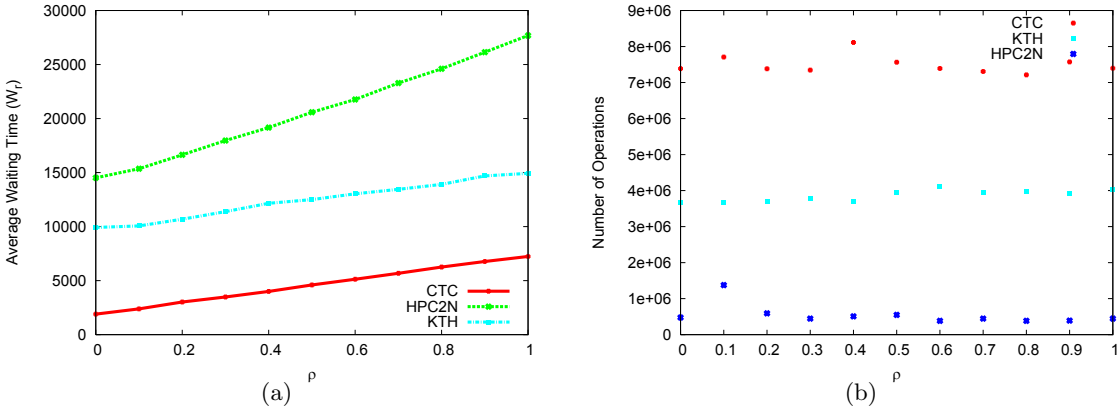

Figure 7: (a) Average waiting time for workloads CTC, KTH and HPC2N, (b) Number of operations as a function of ρ for workloads CTC, KTH and HPC2N

showed that our co-allocation algorithm scales to systems with large number of resources and heavy workloads.

Our co-allocation algorithm has numerous applications, from allocating multiple desktop machines for classroom instruction in a VCL-like computing infrastructure, to scheduling link-wavelength resources in optical Grids, to middleware such as MapReduce that need to allocate compute nodes to handle multiple map and reduce instances.

7. REFERENCES

[1] Google and IBM look to next generation of programmers. `http://www.ibm.com/ibm/ideasfromibm/us/google/index.shtml`.

[2] Hadoop Virtual Cluster Appliance. `http://www.grid-appliance.org/`.

[3] Parallel Workload Archive. `www.cs.huji.ac.il/labs/parallel/workload`.

[4] S. Averitt, M. Bugaev, A. Peeler, H. Schaffer, E. Sills, S. Stein, J. Thompson, and M. Vouk. The virtual computing lab. In *International Conference on Virtual Computing Initiative*, pages 1–16, Research Triangle Park, NC, May 2007.

[5] A. Ballier, E. Caron, D. Epema, and H. Mohamed. Stimulating grid schedulers with deadlines and co-allocation. INRIA Technical Report, January 2006.

[6] T. Beyene, Y. Xin, M. Turabi, and K. Raza. PCE based grid networking. In *IEEE Symposium on Computers and Communications*, pages 769–774, Aveiro, Portugal, July 2007.

[7] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The portable batch scheduler and the Maui scheduler on Linux clusters. In *ALS'00: Proceedings of the 4th. Annual Linux Showcase & Conference, Atlanta*, pages 27–27, Berkeley, CA, US, 2000. USENIX Association.

[8] C. Castillo, G. N. Rouskas, and K. Harfoush. Efficient QoS resource management for heterogeneous Grids. In *22nd. IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, Florida, US, April 2008.

[9] C. Castillo, G. N. Rouskas, and K. Harfoush. On the design of online scheduling algorithms for advance reservations and QoS in Grids. In *21sts IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–10, Long Beach, California, US, April 2007.

[10] K. Czajkowski, I. Foster, and C. Kesselman. Resource co-allocation in computational Grids. In *8th. IEEE International Symposium on High Performance Distributed Computing*, pages 219–228, Redondo Beach, 1999.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] J. Decker and J. Schneider. Heuristic scheduling of grid workflows supporting co-allocation and advance reservation. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 335–342, Washington, DC, US, 2007. IEEE Computer Society.

[13] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, building a virtual data Grid for gravitational wave scientists. In *11th IEEE International Symposium on High Performance Distributed Computing*, pages 225–255, Edinburgh, Scotland, 2002.

[14] C. Dynamic Multi-Terabit Core Optical Networks: Architecture, Protocols and M. (CORONET). `http://www.darpa.mil/STO/solicitations/CORONET/index.htm`.

[15] L. Eyraud-Dubois, G. Mounie, and D. Trystram1. Analysis of scheduling algorithms with reservations. In *IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[16] A. Farrel, J. P. Vasseur, and J. Ash. A path computation element PCE-based architecture. `http://tools.ietf.org/html/rfc4655`, August 2006.

[17] D. G. Feitelson. Packing schemed for gang scheduling. In *Revised Papers from the International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '96)*, 1996.

[18] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, Lectures Notes in Computer Science, Springer, pages 1–16, Cambridge, MA, US, 2004. Springer Berlin / Heidelberg.

[19] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1294 of *Lecture Notes in Computer Science*, pages 1–34, London, UK, April 1997. Springer-Verlag.

[20] Global Grid Forum. Advance Reservations: State of the art. `www.ggf.org`, June 2003.

[21] L. He, S. A. Jarvis, D. P. Spooner, X. Chen, and G. R. Nudd. Dynamic scheduling of parallel jobs with qos demands in multiclusters and grids. In *Proceedigs of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 402–409, London, UK, 2004. Springer-Verlag.

[22] X. He, X. Sun, and G. von Laszewski. Qos guided Min-Min heuristic for grid task scheduling. *Journal of Computer Science and Technology*, 18(4):442–451, 2003.

[23] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, and G. Singh. A comparison of two methods for building astronomical image mosaics on a Grid. In *International Conference on Parallel Processing Workshops (ICPP)*, pages 85–94, Oslo, Norway, June 2005. IEEE Computer Society.

[24] D. A. Lifka. The ANL/IBM SP scheduling system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[25] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A hunter of idle workstations. In *IEEE 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[26] P. Maechling, H. Chalupsky, M. Dougherty, E. Deelman, Y. Gil, S. Gullapalli, V. Gupta, C. Kesselman, J. Kim, G. Mehta, B. Mendenhall, T. Russ, G. Singh, M. Spraragen, G. Staples, and K. Vahi. Simplifying construction of complex workflows for non-expert users of the Southern California Earthquake Center Community modeling environment. *ACM SIGMOD Record*, 34(3):24–30, 2005.

[27] R. Manager Torque. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`.

[28] H. on Demand. `http://hadoop.apache.org/core/docs/r0.16.4/hod.html`.

[29] Phosphorus. `http://www.fz-juelich.de/jsc/grid/PHOSPHORUS/`.

[30] Platform Computing Corporation. LSF. `http://www.platform.com`.

[31] M. K. Ramamurthy and K. K. Droegemeier. Linked Environments for Atmospheric Discovery (LEAD): A cyberinfrstructure for mesoscale meteorology research and education. In *Proceedings of the Conference of Interactive Information Processing Systems for Meteorology, Ocenoegraphy and Hydrology*, 2004.

[32] M. Siddiqui, A. Villazón, and T. Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *ACM/IEEE Conference on Supercomputing*, page 103, New York, NY, US, 2006. ACM.

[33] W. Smith, I. Foster, and V. Taylor. Scheduling with advance reservations. In *14th. IEEE International Parallel and Distributed Processing Symposium*, pages 127–132, Cancun, Mexico, May 2000.

[34] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '02)*, pages 55–71, London, UK, 2002. Springer-Verlag.

[35] D. Talby and D. G. Feitelson. Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, page 513, Washington, DC, US, 1999. IEEE Computer Society.

[36] W. Zhanga, A. M. K. Chengb, and M. Hu. Multisite co-allocation algorithms for computational Grids. In *20st. International Parallel and Distributed Processing Symposium*, pages 1–8, Rhodes Island, Greece, April 2006.