J Supercomput (2015) 71:1297–1317
DOI 10.1007/s11227-014-1361-0

Adaptive multiple-workflow scheduling with task rearrangement

Wei Chen · Young Choon Lee · Alan Fekete · Albert Y. Zomaya

Published online: 13 January 2015

Abstract Large-scale distributed computing systems like grids and more recently clouds are a platform of choice for many resource-intensive applications. Workflow applications account for the majority of these applications, particularly in science and engineering. A workflow application consists of multiple precedence-constrained tasks with data dependencies. Since resources in those systems are shared by many users and applications deployed there are very diverse, scheduling is complicated. Often, the actual execution of applications differs from the original schedule following delays such as those caused by resource contention and other issues in performance prediction. These delays have further impact when running multiple workflow applications due to inter-task dependencies. In this paper, we investigate the problem of scheduling multiple workflow applications concurrently, explicitly taking into account scheduling robustness. We present a dynamic task rearrangement and rescheduling algorithm that exploits the scheduling flexibility from precedence constraints among tasks. The algorithm optimizes resource allocation among multiple workflows, and it often stops the influence of delayed execution passing to subsequent tasks. The experimental results demonstrate that our approach can significantly improve performance in multiple-workflow scheduling.

W. Chen · Y. C. Lee (✉) · A. Fekete · A. Y. Zomaya
School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia
e-mail: young.lee@sydney.edu.au

W. Chen
e-mail: wche4135@uni.sydney.edu.au

A. Fekete
e-mail: alan.fekete@sydney.edu.au

A. Y. Zomaya
e-mail: albert.zomaya@sydney.edu.au

Keywords Scheduling · Workflow applications · Workflow scheduling · Rescheduling

1 Introduction

Large-scale distributed computing systems (LDCSs), such as grids and clouds, have emerged as an essential infrastructure for large-scale and resource-intensive applications, particularly in scientific and engineering domains. An important class of applications, referred to as *workflow*, is characterized by having each job made up of tasks with temporal constraints among the tasks, arising for example because some task needs data or control information produced in earlier tasks of the same job. For example, scientific workflows in bioinformatics [1] and earthquake studies [2], involve the orchestration of serial short-duration tasks in data collecting, processing and transferring. Real-time weather prediction modeling [3] is an application with, in addition, strict deadlines on task completion.

LDCSs allow great amounts of heterogeneous computing resources, which are pervasive in our environment, be harnessed and managed cooperatively for a common objective. LDCSs provide a cost-effective way that uses existing distributed resources to support a range of computing applications. However, the efficiency of these systems has become ever serious with the poor utilization of 10 % or even lower in many cases [4].

There have been many studies on workflow scheduling in LDCSs, e.g., [5–11]. The most popular ones are list scheduling heuristics, which first rank tasks into a priority list based on the data dependency, and then select resources for each of them one after another. A typical example is heterogeneous earliest-finish-time (HEFT) [5], which schedules each task to the resource that can finish it in the earliest time. The approach was first designed mainly for tightly coupled heterogeneous computing environments but it was also applied to loosely coupled systems, such as ASKALON [12]. Several variants of list heuristics were proposed later, such as group (or level based) heuristics [6], and critical path-based heuristics [7,8]. When future tasks of a job are allocated to be run on a resource at a particular time, we say that an *advance reservation* has been made; this is a crucial resource provisioning mechanism adopted by many scheduling algorithms [13–16], especially when QoS guarantees are required [17]. Nevertheless, they mostly deal with a single workflow at a time without paying much attention to performance uncertainties.

In this paper, we address the problem of scheduling multiple, concurrent workflow applications, explicitly taking into account resource efficiency and scheduling robustness. More specifically, we study how to efficiently utilize heterogeneous resources when multiple workflows are to be concurrently scheduled in the system. This scheduling is much more complicated compared to traditional single workflow scheduling due primarily to the fact that resources are heterogeneous and shared. Also, to our best knowledge, there is no existent method to get the accurate performance information on workflows, such as execution times of tasks and data transfer times. These factors have motivated us to design scheduling algorithms that effectively deal with performance uncertainties using rescheduling and task rearrangement techniques.

This paper significantly extends our previous work [18] with the consideration of performance uncertainty. Performance uncertainty is especially damaging in workflow scheduling, since a delay in finishing one task can propagate widely, causing delays among many jobs and on many resources. For example, if a task in a job finishes later than expected, the delay may propagate not only to its successor tasks, but also to tasks of other jobs assigned to those resources of the task and its successor tasks. If nothing is done to reconsider the schedule, the overall execution will be greatly damaged.

We propose a novel approach of reservation adjustment that applies our rescheduling algorithm [18] to deal with delay arising from inaccuracy in performance information. Our approach explicitly takes into account temporal constraints of (scheduled) workflow tasks and it determines whether influenced reservations could be rearranged with tasks of other workflow applications. This rearrangement will allow all tasks run on an appropriate resource instance and thus limiting the delays to scheduled applications.

Extensive experiments have been conducted to study the performance of our scheduling approach under different degrees of prediction accuracy. Results show that our reservation rearrangement algorithm effectively reduces the propagation of postponement among reservations; therefore, scheduling robustness improves.

The rest of this paper is organized as follows. Section 2 describes the problem of multiple-workflow scheduling. In Sect. 3, we present our rescheduling with a graph search algorithm. Section 4 shows how our rescheduling solution is extended to deal with inaccurate performance prediction. Performance evaluation results are presented in Sect. 5 followed by related work in Sect. 6. We then draw our conclusion in Sect. 7.

2 Problem description

In this section, we describe the problem of multiple-workflow scheduling. We first give a brief description of system model and application model, respectively, and then illustrate two specific issues in our problem: resource efficiency and scheduling robustness.

2.1 System model

The target system in this study is viewed as a set of heterogeneous computing/resource sites, each of which consists of a set of resources. Resource sites are loosely coupled, for example through a wide area network with various network bandwidths (as shown in Fig. 1). More formally,

$$S = \{R_1, R_2, \ldots, R_n\}$$

where R_i is an autonomous resource site that can take computing tasks independently. Resources vary in computing capacity (the number of processors, the size of memory and hard disk), computing power (CPU frequency and internal network bandwidth) and other resource properties (such as hardware platform, operating system or services provided). A resource can run several tasks concurrently if the availability and computing capacity allow.

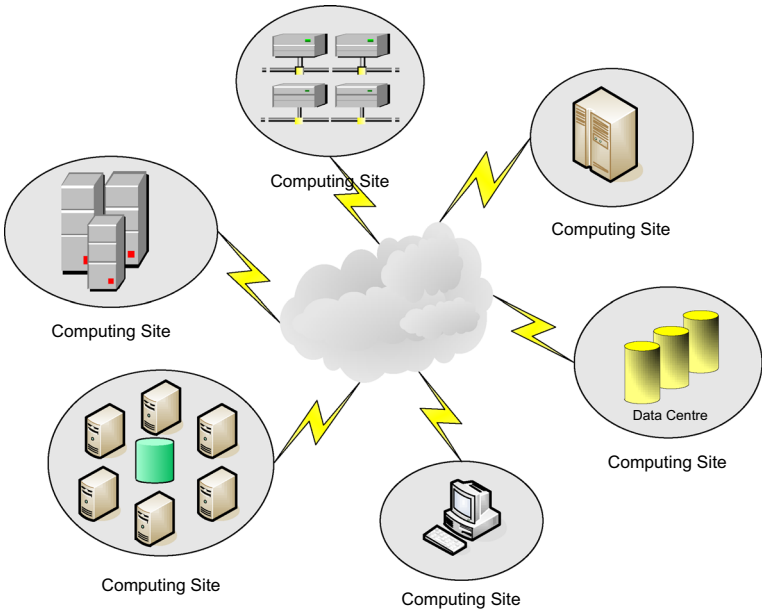

Fig. 1 A large-scale distributed system

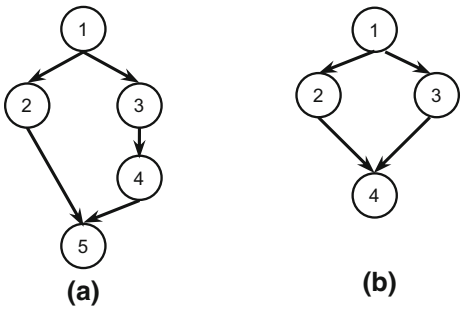

Fig. 2 Two simple DAGs. **a** Job A with five tasks. **b** Job B with four tasks

2.2 Workflow application

A workflow application/job consists of precedence-constrained tasks with data dependencies. The execution of these tasks is coordinated in the way that precedence constraints are respected. A workflow job can be represented by a directed acyclic graph (DAG), $G = (V, E)$, where V is a set of tasks and E is a set of directed edges representing precedence constraints between corresponding tasks. Figure 2 gives two simple examples of DAGs.

For a given edge from task v to task w, v is defined as w's *predecessor*, and w is defined as v's *successor*. A task without any predecessors is defined as *entry task* and a task without any successors is defined as *exit task*. A task is regarded as ready to run (or simply as a 'ready task') when all its predecessor tasks have completed and transmitted the information needed. Thus, the readiness of each task is determined by

its predecessors, more specifically the one that completes the communication at the latest time. More formally, the earliest start time (EST) of a task is defined as:

$$\begin{aligned}\mathrm{EST}(t, r) = \max\{&\mathrm{FT}(t_i) + \mathrm{data}(t_i, t)/\mathrm{bandwidth}(r_i, r) : \ t_i \in \mathrm{predecessor}(t),\\ &r_i = \mathrm{resource}(t_i)\},\end{aligned}$$

where FT is the function giving the expected finish time for each task, and communication (data transmission) time is calculated from the data size and network bandwidth between corresponding resources.

A task with multiple parent tasks requires synchronization. This synchronization means that each task has a latest finish time (LFT) calculated by the need to provide information to each successor in time for that successor to run:

$$\begin{aligned}\mathrm{LFT}(t, r) = \min\{&\mathrm{ST}(t_j) - \mathrm{data}(t, t_j)/\mathrm{bandwidth}(r_j, r) : \ t_j \in \mathrm{successor}(t),\\ &r_j = \mathrm{resource}(t_j)\}\end{aligned}$$

where ST is the expected start time. We assume that the structure of a given workflow is known at the time of job submission, and the execution time of a task can be estimated using performance modeling/prediction techniques, e.g., [20,21].

2.3 Resource efficiency when scheduling multiple workflows

When a new job arrives, the scheduler decides on where (on which resource) and when (at which time) each of its tasks is to be executed; that is, an *advance resource reservation* (or advance reservation for short) is made for each task for the resource of the need to have capacity available at the appropriate period. Note that when one workflow job is scheduled, there are often some tasks of previously submitted jobs waiting for their predecessors to finish. This paper studies cases where previous reservations should be changed as the execution proceeds.

Now, consider two workflow jobs A and B, as shown in Fig. 2, submitted successively to a scheduler for execution over resources R1, R2 and R3. Suppose that from consideration of critical path [7,8] and estimates of duration of each task and inter-task communication needs, the scheduler allocates tasks as shown in Fig. 3a. The system makes advance reservations for these tasks. Then, job B arrives. Due to the resources that have been reserved for job A, task B3 has to be scheduled after task A2. The best scheduling that can be done without changing the reservations for task A is shown by the solid rectangles in Fig. 3b.

However, if task A2 could be rescheduled to a later time slot on another resource, such as the dashed rectangle in Fig. 3b, then task B3 could start earlier and job B could finish earlier. An optimized resource allocation is shown in Fig. 3c. Clearly, although the first schedule of task A2 makes it finish at the earliest time, it becomes a bad choice when job B arrives. A rearrangement of scheduled tasks can optimize the resource allocation for both jobs.

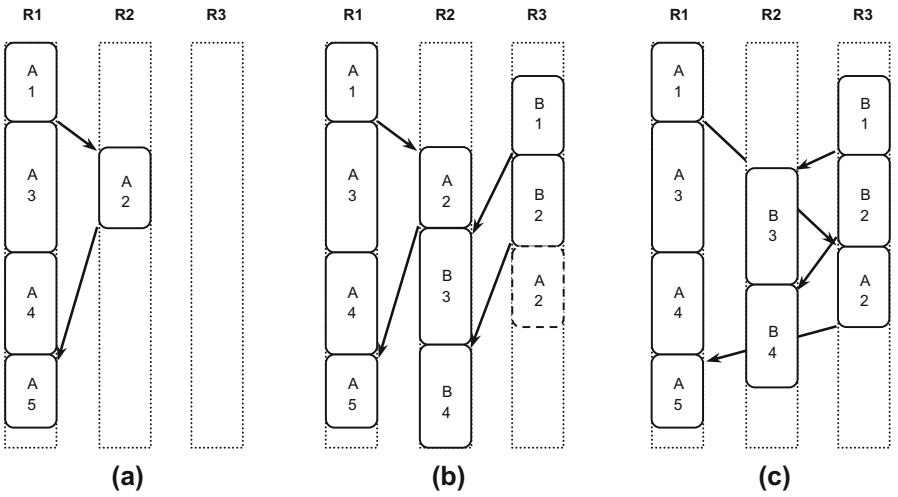

Fig. 3 Task rescheduling to optimize resource allocations. **a** The initial scheduling of job A. **b** Scheduling job B with regard to job A. **c** Optimized scheduling after rescheduling task A2

2.4 Broken reservations and delay propagation

An advance reservation is made based on predictions of task execution time and data transmission time. However, such a reservation may not be always kept as the actual task execution and/or data transfer may differ from the estimation. We describe a reservation that cannot be kept as a *broken reservation*. A broken reservation may propagate to other reservations due to precedence constraints. In other words, for a particular broken reservation, if we simply prolong it until the resource becomes available, other reservations may further need changing. Thus, the broken reservations *must* be adjusted if we are to minimize performance degradation such as an increase of makespan. A more effective way to deal with this issue of broken reservation is reservation rearrangement that considers the other tasks in the system. Figure 4 illustrates our rearrangement scenario.

Suppose there are tasks scheduled on distributed resource sites and the advance reservations are shown as Fig. 4a. Among them, task X1, X2 and X3 belong to a workflow job and need to be executed sequentially. At time t_1, a delay in the execution of X1 is detected. This delay is expected to influence its successor task X2, and the delay may further pass down to more tasks such as Z1 and X3. However, these delays can be avoided if task X2 is rescheduled to resource R3 as shown in Fig. 4b.

3 Rescheduling algorithm

For both issues described above, resource efficiency for multiple workflows and scheduling robustness with broken reservations, we adopt an adaptive rescheduling mechanism to improve performance or reduce the propagation of delays. In this sec-

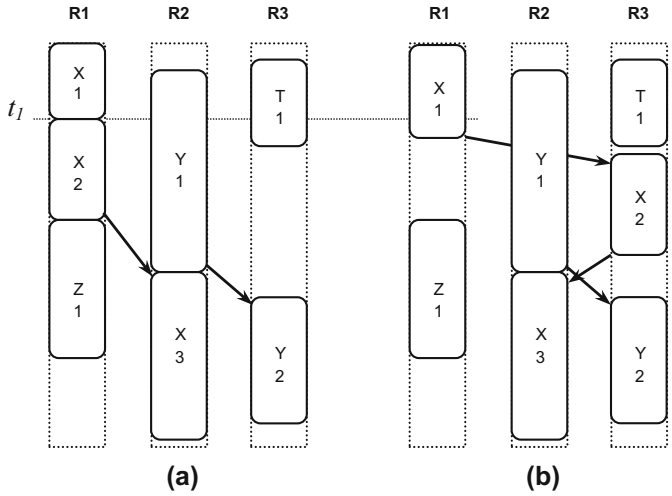

Fig. 4 Task rescheduling to avoid unnecessary postponement

tion, we present our rescheduling algorithm. Then, in the next section, we will extend our algorithm with the consideration of broken reservations.

3.1 Task-resource matching

If we take tasks and resources as two disjoint sets, then a scheduling problem becomes a bipartite matching problem.

For example in Fig. 5a, nodes in part T represent tasks (each belongs to a particular workflow) and nodes in part R represent resources. For a given task, a resource is defined to be *satisfiable* if it meets the task's resource requirements including computing and storage capacities. We make every task linked with all its satisfiable resources in the bipartite graph. So, T1 has one satisfiable resource: R1, and T2 has two: R1 and R2. An arrow on the line pointing to the task represents a case where the resource is actually allocated to that task (that is, the appropriate amount of CPUs and memory is reserved for the task for a specific period of time); the arrow points away from the task otherwise. In the figure, it shows that T1 and T2 are scheduled on R1 and R2, respectively, and, T3 is not scheduled at the moment.

To schedule T3, we need to check all its satisfiable resources (R2 in our example) to get an appropriate time slot during which available resources can be reserved to meet T3's requirements. However, if R2 is already too committed for the time period involved, then we cannot schedule T3 without further changes. Thus, the workflow to which T3 belongs would need to be either rejected for the execution or the deadline would be relaxed. Instead, we propose to try to rearrange existing reservations so that we can place T3 on R2. We scrutinize reservations on R2 if any task can be rescheduled to satisfy T3's temporal constraint. In Fig. 5a, we assume that T2 can be removed from R2 for this purpose. Then, the problem becomes whether T2 can be rescheduled successfully on another resource instance or time slot without violating

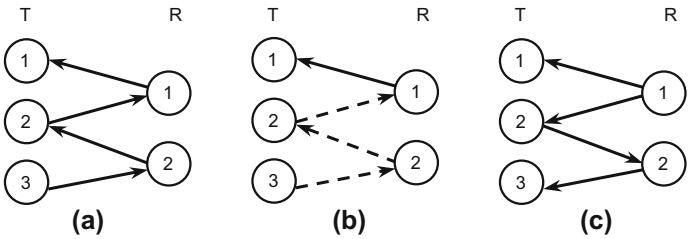

Fig. 5 An example of resource allocation rearrangement

its temporal constraint. In our simple case, we assume that R1 has enough resource for both T1 and T2 and then no more tasks need be removed. This path of rescheduling tasks can be shown as the dash lines in Fig. 5b. By reversing the direction of these dash lines, we get a new matching in Fig. 5c, where T3 is scheduled on R2 and T2 shares resource R1 with T1.

3.2 Dynamic search tree

A traditional bipartite graph matching algorithm [22] can find the maximal one-to-one matching in a bipartite graph. However our problem is more complicated. First, task scheduling involves many-to-one matching. For example, T2 can share resource R1 with T1. Second, we cannot simply determine whether a task can be matched with (or scheduled on) a resource in advance: temporal constraints and resource availability must be considered while these depend on the scheduling of other tasks. Any task rescheduling will leave different resource availability to others and different temporal constraints to its predecessors and successors.

We develop a new match searching algorithm based on a dynamic search tree. Our algorithm adopts the basic ideas from the standard bipartite matching algorithm: rearranging the matches along an *augmenting path*, which alternates edges that are used in the matching and that are not used (as the dash lines shown in Fig. 5b). In searching for the augmenting path, each step along an edge means that a task is assigned to a resource by removing another task from it. The removed task then triggers the next step of replacement. Theoretically, we can remove several tasks from a resource to make enough room for others. To simplify the computation, we restrict that only one task is replaced by another in each step of rearrangement that we consider. We define a tuple:

$$\langle t1, r, t2 \rangle$$

to represent where a task $t1$ can be scheduled on one of its satisfiable resources r by replacing another task $t2$ that is currently scheduled to r. We call $t1$ the removed task (it is removed from scheduling in the last replacement), $t2$ the replaceable task (it will be the removed task in the next step), and r the satisfiable resource. For any removed task, there may be many possibilities of such replacement through different satisfiable resources and alternative tasks on a resource. The algorithm goes

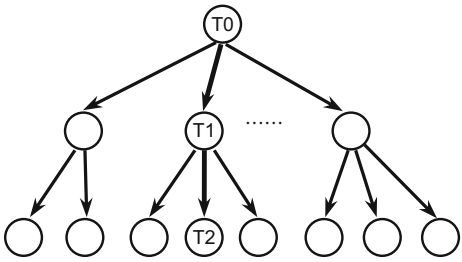

Fig. 6 A search tree for augmenting path (a serial of rearrangements)

through each of them until any task can be rescheduled without replacing any other tasks.

Our search begins from the task that needs to be scheduled. It can be regarded as a search tree (as shown in Fig. 6) that grows down from removed tasks to their children: replaceable tasks. The replaceable tasks then become removed tasks and the search tree extends to one lower level of replaceable tasks. A search path along the tree represents a sequence of rearrangements in matching. For example, the bold lines in the figure represent that T0 can be scheduled by replacing T1 on the resource where T1 is currently matched, and T1 can be rescheduled by replacing T2 on the resource where T2 is currently matched. If T2 is rescheduled without replacing any other task, this path will be an augmenting path that we are searching for.

The search tree is built up dynamically during the search process. When the search reaches a task node along a path, we calculate the current status of resource availability (after a sequence of rearrangements among the concerned resources on the path) and the task's time slot boundary according to the current scheduling of its predecessors and successors. Then, we determine whether the task can be rescheduled on an available resource, if not, we determine which tasks can be replaced in the next step to make room. The search tree is growing level by level as the search goes deep. When it reaches the deepest task along a branch, the search tracks back to the upper level. At the moment, the rearrangement along the retreated path must also be rolled back, and then search continues after a new level of task nodes is calculated out along another branch.

3.3 Search algorithm

Our rescheduling algorithm (Fig. 7) adopts a depth-first-search in the process to build the search tree and to look for an augmenting path. In each step of search, the algorithm first checks whether it is going deeper along a path or backtracking to an upper level (line 7). If it is backtracking, the rearrangements along the retreated path are also rolled back (lines 9 and 10) to restore the former scheduling status. Search path goes forward by task replacing in line 12, and new scheduling is calculated according to the current status of task-resource matching.

In our algorithm, we restrict the reschedulings we offer for a task to those that are within its time slot boundary (between EST and LFT), so that we avoid rescheduling the entire workflow (more precisely, predecessors and successors of the task). For

```
NOTES:
rearrangement: tuple<t1, r, t2>, refer to section 3.2
top(S): the top element in stack S
task(r): all tasks scheduled on resource r
resource(t): all satisfiable resources of task t
schedulable(t, r):
  check whether task t can be scheduled on resource r
  within time slot boundary EST(t, r) and LFT(t, r), refer to section 2.2
schedulable(t, r, t'):
  check whether task t can be scheduled on resource r
  by removing task t'

Input a task t needs to be scheduled
Output augmenting path (new scheduling of tasks)

1.  T = an empty stack of rearrangement   // dynamic search tree
2.  P = an empty stack of rearrangement   // augmenting path
3.  push(T, <null, null, t>)   // initialize search tree with inputted task

4.  while (T is not empty) {
5.    x = pop(T)
6.    if (x.t1 != null) {      // if x is not root node
7.      while (x.t1 != top(P).t2) {       // backtracking?
8.        // if backtracking, roll back rearrangement along retreated path
9.        y = pop(P)
10.       y.t2 replace y.t1 on y.r
11.     }
12.     x.t1 replace x.t2 on x.r          // make rearrangement
13.   }
14.   push(P, x)                // extend augmenting path
15.   for (each r in resource(x.t2)) {
16.     if (schedulable(x.t2, r)) {
17.       push(P, <x.t2, r, null>)
18.       return P
19.     }
20.     else {
21.       for (each t' in task(r)) {
22.         if (schedulable(x.t2, r, t') push(T, <x.t2, r, t'>)
23.       }
24.     }
25.   }
26. }
27. return scheduling failed
```

Fig. 7 Task rescheduling algorithm

example, the rescheduling of task A2 in Fig. 3 does not require task A1 or A5 to also be rescheduled for the temporal constraints.

We adopt a heuristic to trim the search tree so that the time complexity is affordable. We restrict that each task appears at most once in the search tree. Thus, in the worst case, our search goes through n nodes, where n is the number of tasks scheduled in the system. In each step of search, all the satisfiable resources of a task are checked for rescheduling; hence, the time complexity is $O(sn)$, where s is the average number of satisfiable resources for each task.

4 Saving broken reservations

Advance reservation is an important resource provisioning mechanism widely adopted in many scheduling algorithms [13–17]. Our algorithm makes resource reservations

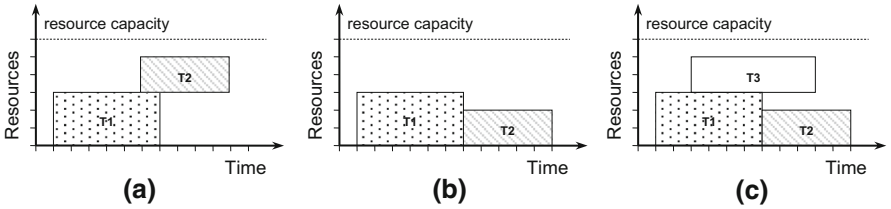

Fig. 8 An example of task assignment in a resource site. **a** Resource reservations for task T1 and T2. **b** Reservation for task T2 is postponed. **c** Resource allocation if task T3 is rescheduled on the resource

for scheduled tasks that may belong to multiple workflows and improves resource efficiency by rearranging reservations (or in other words, rescheduling) when new tasks arrive. However, the choice of advance reservation is based on the prediction of task execution time and data transmission time, which are often not accurate. Broken reservations are rather common and should be explicitly dealt with to keep scheduling consistency in the system.

4.1 Reservation compatibility in rescheduling

Inaccurate performance prediction, specifically underestimated running time, will produce delays in task execution. These unexpected delays may use resources that have been reserved for other tasks and, therefore, break those reservations. One possible approach to resolve the problem could be by postponing the start time of all broken reservations (including those broken by broken reservations) by a fixed time, for example, introducing one hour postponement if delay is one hour. Rather, we advocate the rescheduling of the task, as described in Sect. 2.4, to stop postponement propagating to more and more reservations.

There is one issue that must be considered when rescheduling to save broken reservations: the rescheduled task must not induce any extra postponement to other reservations, because that may offset the benefit we obtain from rescheduling. We illustrate the issue with an example in Fig. 8.

Consider a resource instance which is reserved for two tasks as shown in Fig. 8a. The reservations are compatible as the total required computation capacity, such as the size of memory and the number of processors, meets the overall capacity of the resource instance. Now suppose that a delay in execution breaks a list of reservations among these being the reservation for task T2. To save these reservations, we try to reschedule corresponding tasks. However, suppose that rescheduling of T2 failed and the reservation has to be postponed for a certain time, as shown in Fig. 8b. Then, we continue to reschedule other tasks in the list and our algorithm may find that task T3 could be rescheduled on the resource as shown in Fig. 8c. Although the reservations stay fit within the capacity, it could become a problem if reservation for task T1 would be postponed after this rescheduling: that would force T2's reservation to be postponed again. This is because the rescheduled task T3 used the resource capacity that originally can run tasks T1 and T2 concurrently. Delays are quite possible from

inaccurate performance prediction. We must avoid excessive postponement occurring to one reservation because it compromises the benefit we obtain from rescheduling.

To avoid such situations, we revise our algorithm in checking whether a rescheduled task is compatible with all others on the resource instance. We first hypothetically postpone all un-postponed reservations in the concerned time period and calculate whether the resource allocation is still compatible. In the above example of Fig. 8, we first hypothetically postpone task T1 by one unit of time, and then we will find that the resource allocation would exceed the capacity if task T3 would be rescheduled on it. So, we consider that rescheduling T3 as illustrated would not be compatible, and we must search for another resource instance or time slot for task T3 in the rescheduling. If the resource allocation is compatible when all un-postponed reservations are hypothetically postponed, it must be compatible when arbitrary reservations are really postponed after the rescheduling.

4.2 Algorithm description

Our rescheduling algorithm is extended to deal with broken reservations. We adopt periodic polling to monitor the actual progress in task execution and data transmission. While earlier completion of task execution and data transmission can be easily dealt with existing techniques, such as back-fill algorithms [23,24], our focus is the counter case when such completion is (being) delayed, particularly affecting other reservations.

At a polling point, if a delay in task execution is detected we need to extend the length of reservation; if a delay in data transmission is detected we have to postpone the start time of the task. We set the time length (in extending or postponing reservation) equal to the interval of two polling points. If the delay is not over within that time, it will be detected and adjusted again at the next polling point. A higher polling rate would make these adjustments in task finish/start time more accurate, but with the expense of higher overhead on rescheduling.

After detecting all delays in task start and finish time, we calculate which reservations are influenced by these delays. The influenced reservations are collected into two lists: one (PList) is for those that can only be postponed, such as the ones that have received data at the resource site (we do not consider to re-transfer data in rescheduling), and the other (RList) is for those that could be rescheduled to a different resource. We use these two lists in our algorithm, shown in Fig. 9.

Reservation adjustment begins from the first list (PList). The new allocation of each postponed reservation may further influence others, on the same resource site or among the postponed task's successors. These influenced reservations are appended to the lists based on whether they can be rescheduled or not. When all the reservations that have to be postponed have been adjusted, the algorithm begins to reschedule the reservations in the other list.

We use the algorithm described in Sect. 3, with stricter checking on reservation compatibility as discussed in Sect. 4.1. If rescheduling of a task fails, its original reservation will be postponed. The process continues until all reservations in both lists are adjusted appropriately.

```
Input PList = reservations have to be postponed
      RList = reservations could be rescheduled

while (PList or RList is not empty) {
  for (each reservation in PList) {
    postpone reservation by one unit of time
    if (postponement breaks other reservations)
      add them to PList or RList
  }
  for (each reservation in RList) {
    Reschedule reservation
    if (rescheduling fails) {
      add reservation to PList
      break
    }
  }
}
```

Fig. 9 Reservation adjustment

Table 1 Experimental setup

Resource		Workflow	
Setting	Normalized value range	Parameter	Range of values
Computing capacity	1–10	$\|V\|$	20–100
Computing speed	1–8	CCR	0.1–10
Network bandwidth	1–8	α	0.5–2.0

5 Performance evaluation

We have conducted extensive simulation experiments to evaluate the efficacy of our algorithm. This section details experimental settings and scheduling scenarios, and presents results.

5.1 Experimental settings

In this section, we describe characteristics of computing resources and workflow jobs followed by the model we use for the inaccuracy of performance prediction.

5.1.1 Computing resources

We have simulated a virtual system consisting of 1,000 heterogeneous resource sites. Each resource is configured with different settings to simulate necessary computation features that are concerned in task scheduling. In particular, we virtualize each resource site as a tuple (*computing capacity*, *computing power*, *network bandwidth*); these elements are assigned with normalized values (Table 1).

Computing capacity is a parameter that describes the volume of a resource site. It is usually represented by the number of processors (CPUs or Cores) and the size of memory. We assume that at any time a certain number of processors and memory blocks are exclusively allocated to a task. Therefore, computing capacity determines how many tasks (each has a specific requirement on the number of processors and the size of memory etc.) can concurrently run on a resource site.

Computing speed represents the comprehensive performance of a resource site. It is often reflected by the frequency of CPU, bandwidth of data bus and I/O throughput of memory and disk. This parameter is used to determine the execution time of a task on a given resource site.

Network bandwidth defines the performance of external network that connects resource sites. Our target system simulates resource sites that are linked through Internet and, therefore, they are fully connected but the bandwidths vary among the links. In simulations, each resource site is assigned with a bandwidth value but the bandwidth of a link is the minimal value of the two resource sites. Network bandwidth determines transmission time when data are transferred from one resource site to another.

There are other resource properties that may be considered in scheduling. For example, tasks may require specific operating system, software or hardware devices be installed on the target resource. As defined in Sect. 3.1, these properties determine the satisfiable resources of a task. In our experiments, we configure properties of resource sites and task requirements in the way that the number of satisfiable resources for each task is in a Poisson distribution, and on average each task has 50 % of total resource sites in the system as its satisfiable resources.

5.1.2 Workflow jobs

Each workflow (DAG) is produced with varying parameters including the total number of tasks $|V|$, the communication to computation ratio CCR and shape factor α (that reflects the parallelism degree of a job). The value range of these parameters in our experiment is summarized in Table 1.

We randomly select each parameter value from the appropriate range when producing workflow jobs. Therefore, the jobs are various in size and parallelism degree. Some workflows are compute intensive and others are communication intensive.

We measure the performance of our algorithm when workflow jobs arrive sequentially; however, one may arrive before the previous ones have completed, and thus workflow jobs run concurrently in the system. The number of jobs arriving in a unit of time is Poisson distribution. By giving different arriving intervals (the mean value of the distribution), we compare the performance of scheduling algorithm under different pressure of resource contention. We use an abstract time unit in calculating workflow makespan and job arriving interval. In practice, this unit could be hour or minute, but this does not affect the experimental results.

Each task in a workflow job has a specific requirement on the number of processors and the size of memory. We normalize this as a number between one and eight to simulate various requirements on computing capacity. In practice, this requirement can be determined in advance according to the internal structure of task (parallel processes/threads, data buffer usage, etc.). We assign a number *workload* to each task

that determines its execution time on a given resource site (as workload divided by computing speed). The length of task execution time distributes from 10 to 800 (units of time) in our experiments. A task may also require input data from its predecessors. We assume that the data size is known when scheduling and the transmission time is calculated from the network bandwidth between the two resource sites. There are also local jobs arriving randomly in each resource site. They are scheduled on the earliest time slots available on the resource site, but they cannot preempt scheduled workflow tasks. In our experiments, we set local jobs to take 30 % of resource capacity.

5.1.3 Inaccuracy in prediction

To model the inaccuracy in performance predictions, we introduce an inaccuracy ratio as: (*predicted* − *actual*)/*actual*. We let the value of this inaccuracy ratio be a normal distribution (with $\mu = 0$), both for predicting task execution time and data transmission time. In the meantime, we consider experiments with different standard deviations of the distribution, to simulate different degrees of prediction inaccuracy; the standard deviations we use are 0.1, 0.15 and 0.2.

Given the heterogeneous characteristics of the system, the predictions of the same task on different resource sites would have independently chosen inaccuracy ratios. Based on all these potentially inaccurate predictions, we make an advance reservation for each scheduled task that forms part of a newly submitted job.

5.2 Scheduling scenarios

We consider two different scenarios of multiple-workflow scheduling in our evaluation study. The distinction is based on whether workflow jobs have deadlines. The main objective when scheduling workflows with deadline is to accommodate more jobs (high throughput), whereas the objective of scheduling workflow jobs without deadline is to minimize the completion time (low makespan). Our rescheduling approach helps improve the scheduling performance for multiple workflow jobs in both scenarios.

5.2.1 Scenario 1: rescheduling for high throughput meeting deadlines

In this scheduling scenario, each workflow job comes with a deadline. That may be a requirement from users or an agreement between users and resource providers. In the experiments, we set a deadline for each workflow job according to task execution time on the average computing speed. By adjusting workflow deadlines, we intentionally make some jobs more urgent than others.

We guarantee a workflow job's deadline by scheduling each of its tasks within a task deadline. The task deadline is calculated according to the workload proportions of tasks and edges. During the scheduling, each task is assigned to a resource instance in which it can be finished before the deadline. This assignment may need to rearrange previously scheduled tasks to leave enough computing capacity for the newly arrived ones. But if this deadline-based scheduling fails, our approach will turn to the traditional greedy algorithm that schedules each task to achieve the earliest finish time.

The calculated earliest finish time (EFT) may lie beyond the deadline of the job. In that case, the job has to be rejected. We have discussed the details of the scheduling approach in [18].

5.2.2 Scenario 2: rescheduling for low makespan

In this scenario, we schedule each task in the EFT to minimize makespan. We employ our task rearrangement approach for a new task to finish earlier without influencing the makespan of other scheduled jobs.

When a task is to be scheduled, our approach first calculates the EFT it could have, based on the current resource reservations (or the current schedule). Our approach further calculates whether it could be finished even earlier than the initial EFT if any other task would be removed from the schedule. We collect a list of such tasks: by removing any of them, the task under scheduling can be finished earlier than its 'earliest finish time'. Then, our rescheduling algorithm is employed to check whether any of these removed tasks can be rescheduled successfully. For exit tasks, the rescheduling must not extend the makespan of the job. If the rescheduling fails on all candidate tasks, the task under scheduling will be scheduled on the EFT calculated in the first step.

5.3 Experimental results

Our experiments start by evaluation of four existing workflow scheduling algorithms: HEFT [5], DCP-G [7], DAGMap [8] and meta-heuristic GRASP [9]. In particular, we compared the latter three against HEFT. We chose HEFT as a reference algorithm due to its simplicity and performance. We apply advance reservation to these algorithms for scheduling multiple workflows. For each algorithm, the average makespan is measured when a certain number of jobs are concurrently scheduled in the system. Our experiment covers situations from high resource availability to intense resource contention.

We calculate the makespan speedup ratio of each algorithm (DCP-G, DAGMap and GRASP) to HEFT. Figure 10 shows that although these efforts can shorten makespan of workflows (compared against HEFT), the performance advantage degrades quickly in multiple-workflow scheduling. Their performance is quite similar as that of HEFT under high resource contention among workflows. Therefore, we will compare the performance of our solution (DGR) against that of HEFT in different scenarios.

5.3.1 Performance in guaranteeing workflow deadline

In this series of experiments, each workflow job comes with a deadline, and the scheduler needs to ensure that accepted jobs are finished before their deadlines. For the purpose of experiment, we make some jobs be more urgent in execution than others. We introduce a parameter: *urgency* in producing workflow jobs. It is a ratio: (*deadline* − *makespan*)/*makespan*, where predicted makespan is calculated on the average computation/communication speed of the target system. The value of this parameter is chosen from a standard normal distribution among all submitted workflow jobs.

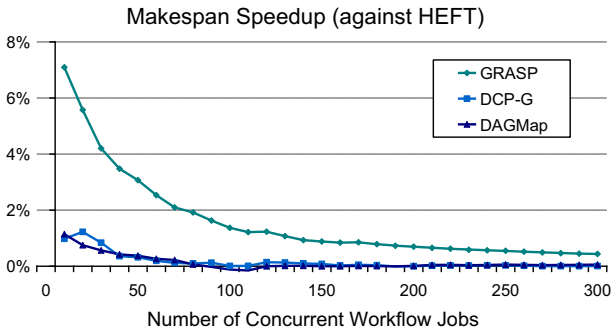

Fig. 10 Performance under multiple-workflow scheduling

As each submitted workflow job is constrained by its deadline, resources are shared and their capacity is limited; the admission of the job is determined based on the "schedulability" of that job. In an ideal situation, the predictions on task execution time and data transmission time are accurate and all accepted workflow jobs will be finished before their deadlines as expected. However, in practice, the inaccuracy in predictions breaks this premise. One way to deal with this QoS degradation is overestimation (or pessimistic estimation with some padded value). Here, the predicted time is more likely to be longer than the actual one for the probability of delay to be reduced. This method has a side effect on the scheduling performance: it reduces acceptance rate and resource utilization, because a larger time slot for each task is reserved.

In the experiment, we apply overestimation according to the degrees of prediction inaccuracy (that were chosen with standard deviation 0.1, 0.15 or 0.2) to keep the deadline satisfiability (percentage of accepted jobs that really finish before their deadlines) at more than 95 %. Under such a situation (achieving the same deadline satisfiability), the performance of our algorithm DGR is compared against HEFT in acceptance rate and resource utilization. We also measure the running time of different algorithms for scheduling and rescheduling (handling broken reservation).

As can be seen in Fig. 11, the performance of DGR is more robust in terms both of acceptance rate and resource utilization than HEFT. In particular, DGR obtains 10–20 % higher acceptance rate and resource utilization than HEFT. With the consideration of workflow job running time, milliseconds' increasing in scheduling and rescheduling running time is ignorable. It takes less time when prediction inaccuracy is high, because more overestimation is required in that case to obtain the same deadline satisfiability. The overestimation relaxes the resource contention in scheduling.

5.4 Performance in minimizing workflow makespan

In this set of experiments, all the submitted jobs are accepted and scheduled in the minimal makespan. We measure the average makespan of workflow jobs scheduled by our algorithm and HEFT when the resource contention scales up.

Figure 12a shows the average makespan of jobs scheduled by our algorithm under different predication inaccuracy levels. Although the inaccuracy in predictions

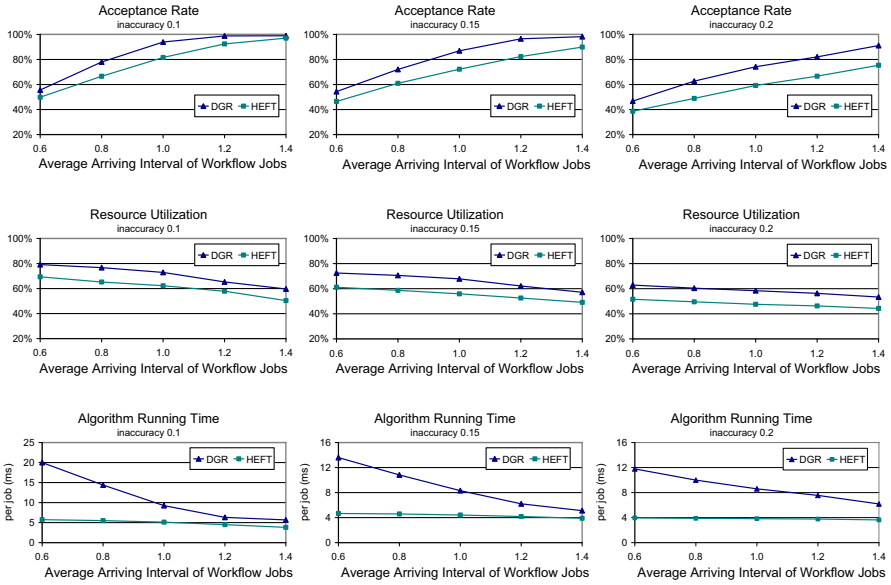

Fig. 11 Performance comparison between DGR and HEFT under different degrees of prediction inaccuracy

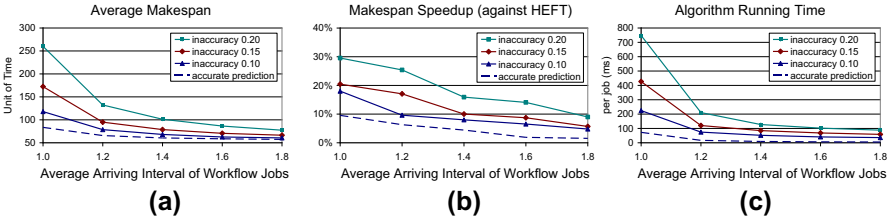

Fig. 12 Performance of our algorithm in minimizing workflow makespan

increases makespan, the impact of such inaccuracy on our algorithm is relatively less than HEFT, since our algorithm delivers a higher degree of speedup as prediction inaccuracy increases (as shown in Fig. 12b). Figure 12c gives the average running time of our algorithm. As no overestimation is applied in this scenario, much time is spent on saving broken reservations, especially when prediction inaccuracy is high. This scheduling overhead can be easily justified by performance benefits (makespan) and the fact that workflow jobs may run hours or days.

6 Related work

Although there have been extensive studies on workflow scheduling, only a small number of them explicitly deal with scheduling of multiple, concurrent workflows (e.g., [13,25–28]). Zhao and Sakellariou [25] convert the problem of multi-workflow scheduling into a single workflow scheduling problem by merging several workflows. However, this approach becomes less effective when workflows constantly arrive due

to its application to static scheduling. Yu and Shi [26] proposed an approach that dynamically prioritizes all ready tasks of different workflows. In particular, the work in [26] schedules workflows in a best effort manner without considering QoS requirements, such as workflow deadline. As the approach in [26] prioritizes small jobs, it is subject to starvation if there are many small jobs arriving constantly. Decker and Schneider [13] apply HEFT to multiple-workflow scheduling with advance reservations; this is the reference algorithm in this paper. Mao and Humphrey [27] deal with multiple workflows constantly arriving in some pattern, stable, cyclic/bursting, growing and on-and-off. The work uses a load vector to consolidate tasks to minimize #resources, or instances in Amazon EC2. The instance acquisition and release are dynamically done to minimize cost without breaching application deadlines. Malawski et al. [28] study the efficiency of executing multiple scientific workflows (termed "workflow ensembles") on public clouds under cost and deadline constraints. The authors developed a set of algorithms to increase the cost to performance ratio and evaluated the effectiveness of these algorithms with a broad range of budget and deadline constraints. These algorithms, however, are not explicitly designed to deal with inaccuracy in performance prediction.

The prediction accuracy of task execution time and data transmission time is vital for scheduling performance and QoS support. Extensive efforts have been made on this matter including [29,30]. Ali et al. [29] mathematically described a general procedure that first identifies performance features and perturbation parameters, and then analyses the impact of perturbation parameters on the performance features. In a later study in [30], they further developed a mathematical model to compute whether a resource allocation can probabilistically guarantee a given level of QoS. Despite these results, accurate prediction is still very rare in practice.

Shi et al. [31] investigated a bi-objective scheduling problem aiming to minimize the makespan and maximize the robustness for workflow schedules. Their study is based on an intuition that the expected makespan of a workflow job is stable if tasks are scheduled with some slacks as cushion to prediction errors. As achieving schedule robustness with slacks and minimizing makespan are two conflicting objectives, they developed a genetic algorithm, by which users can search for a better balance between the two performance metrics.

There are also studies that considered rescheduling in responding to the uncertainty of Grid environment. The work in [11,32] adopts a policy that recalculates the schedules of all unexecuted tasks whenever the execution delay will affect the makespan of the whole job. The studies in [33,34] proposed a rescheduling strategy for workflow jobs when a new resource is discovered or the performance of resource changes. Unlike our DGR algorithm, these approaches reschedule all unexecuted tasks whenever a change in the system occurs. This is less appealing for multiple-workflow scheduling, especially when resource contention is high.

7 Conclusion

In this paper, we have addressed multiple-workflow scheduling incorporating an efficient rescheduling heuristic with the support of task rearrangement. The schedul-

ing of workflow applications is complicated by precedence constraints of tasks and the dynamic and heterogeneous nature of distributed resources across multiple sites. Resource contention among concurrent workflows makes this scheduling problem even more difficult hindering the effective utilization of resources. Our rescheduling mechanism exploits scheduling flexibility of workflow applications to optimize resource allocation. Our study has explicitly considered the uncertainty in resource performance prediction that significantly impacts scheduling robustness. Evaluation showed scheduling performance is greatly improved by our rescheduling approach even under a range of degrees of prediction inaccuracy.

Acknowledgments The work of A. Zomaya is supported by the Australian Research Council Discovery Grant DP1097111. This paper revises and incorporates material from [18,19].

References

1. Oinn T et al (2004) Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17):3045–3054
2. Maechling P et al (2005) Simplifying construction of complex workflows for non-expert users of the Southern California Earthquake Center community modeling environment. ACM SIGMOD Rec 34(3):24–30
3. Plale B et al (2005) Towards dynamically adaptive weather analysis and forecasting in LEAD. In: Proceedings of int'l conf. on computational science, workshop on dynamic data driven applications, pp 624–631
4. Greenberg A, Hamilton J, Maltz DA, Patel P (2008) The cost of a cloud: research problems in data center networks. ACM SIGCOMM Comput Commun Rev 39(1):68–73
5. Topcuoglu H, Hariri S, Wu M (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
6. Mandal A et al (2005) Scheduling strategies for mapping application workflows onto the grid. In: Proceedings of IEEE int'l symp. on high performance distributed computing, pp 125–134
7. Rahman M, Venugopal S, Buyya R (2007) A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In: Proceedings of IEEE int'l conf. on e-science and grid computing, pp 35–42
8. Cao H, Jin H, Wu X, Wu S, Shi X (2008) DAGMap: efficient scheduling for DAG grid workflow job. In: Proceedings of ACM/IEEE int'l conf. on grid computing, pp 17–24
9. Blythe J et al (2005) Task scheduling strategies for workflow-based applications in grids. In: Proceedings of IEEE int'l symp. on cluster computing and the grid, pp 759–767
10. Yu J, Kirley M, Buyya R (2007) Multi-objective planning for workflow execution on grids. In: Proceedings of IEEE/ACM int'l conf. on grid computing, pp 10–17
11. Lee YC, Subrata R, Zomaya AY (2009) On the performance of a dual-objective optimization model for workflow applications on grid platforms. IEEE Trans Parallel Distrib Syst 20(9):1273–1284
12. Fahringer T et al (2005) ASKALON: a tool set for cluster and grid computing. Concurr Comput Pract Exp 17(2–4):143–169
13. Decker J, Schneider J (2007) Heuristic scheduling of grid workflows supporting co-allocation and advance reservation. In: Proceedings of IEEE int'l symp. on cluster computing and the grid, pp 335–342
14. Wieczorek M et al (2006) Applying Advance reservation to increase predictability of workflow execution on the grid. In: Proceedings of IEEE international conference on e-science and grid computing, pp 82–90
15. Smith W, Foster I, Taylor V (2000) Scheduling with advanced reservations. In: Proceedings of IEEE international symposium on parallel and distributed processing, pp 127–132
16. Curino C et al (2014) Reservation-based scheduling: if you're late don't blame us!. In: Proceedings of ACM symp. on cloud, computing, pp 1–14

17. Majumdar S (2009) The any-schedulability criterion for providing QoS guarantees through advance reservation requests. In: Proceedings of IEEE/ACM international symposium on cluster computing and the grid, pp 490–495
18. Chen W, Fekete A, Lee YC (2010) Exploiting Deadline Flexibility in Grid Workflow Rescheduling" deadline flexibility in grid workflow rescheduling. In: Proceedings of ACM/IEEE int'l conf. on grid computing, pp 105–112
19. Chen W (2012) High performance multiple-workflow scheduling using task rearrangement. PhD thesis, University of Sydney
20. Smith W, Foster I, Taylor V (2004) Predicting application run times with historical information. J Parallel Distrib Comput 64(9):1007–1016
21. Duan R, Nadeem F, Wang J, Zhang Y, Prodan R, Fahringer T (2009) A hybrid intelligent method for performance modeling and prediction of workflow activities in grids. In: Proceedings of IEEE int'l symp. on cluster computing and the grid, pp 339–347
22. Kleinberg J, Tardos E (2006) Algorithm design. Pearson/Addison-Wesley, USA
23. Mu'alem AW, Feitelson DG (2001) Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans Parallel Distrib Syst 12(6):529–543
24. Netto MAS, Buyya R (2008) Rescheduling co-allocation requests based on flexible advance reservations and processor remapping. In: Proceedings of IEEE/ACM int'l conf. on grid computing,break pp 144–151
25. Zhao H, Sakellariou R (2006) Scheduling multiple DAGs onto heterogeneous systems. In: Proceedings of the 15th heterogeneous computing workshop
26. Yu Z, Shi W (2008) A planner-guided scheduling strategy for multiple workflow applications. In: Proceedings of int'l conf. on parallel processing-workshops, pp 1–8
27. Mao M, Humphrey M (2011) Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proceedings of 2011 int'l conf. for high performance computing, networking, storage and analysis (SC), pp 49:1–49:12
28. Malawski M, Juve G, Deelman E, Nabrzyski J (2012) Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In: Proc. of int'l conf. on high performance computing, networking, storage and analysis, p 22
29. Ali S, Maciejewski AA, Siegel HJ, Kim J-K (2004) Measuring the robustness of a resource allocation. IEEE Trans Parallel Distrib Syst 15(7):630–641
30. Shestak V, Smith J, Maciejewski AA, Siegel HJ (2008) Stochastic robustness metric and its use for static resource allocations. J Parallel Distrib Comput 68(8):1157–1173
31. Shi Z, Jeannot E, Dongarra JJ (2006) Robust task scheduling in non-deterministic heterogeneous computing systems. In: Proceedings of IEEE int'l conf. on cluster computing, pp 1–10
32. Sakellariou R, Zhao H (2004) A low-cost rescheduling policy for efficient mapping of workflows on grid systems. Sci Program 12(4):253–262
33. Yu Z, Shi W (2007) An adaptive rescheduling strategy for grid workflow applications. In: Proc. of IEEE int'l symp. on parallel and distributed processing
34. Zhang Y, Koelbel C, Cooper K (2009) Hybrid re-scheduling mechanisms for workflow applications on multi-cluster. In: Proc. of IEEE int'l symp. on cluster computing and the grid, pp 116–123