Advance Reservation Policies for Workflows

Henan Zhao and Rizos Sakellariou

School of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, UK
{hzhao,rizos}@cs.man.ac.uk

Abstract. Advance reservation of resources has been suggested as a means to provide a certain level of support that meets user expectations with respect to specific job start times in parallel systems. Those expectations may relate to a single job application or an application that consists of a collection of dependent jobs; in the context of Grid computing, applications consisting of dependent tasks become increasingly important, usually known as workflows. This paper focuses on the problem of planning advance reservations for individual tasks of workflow-type of applications when the user specifies a requirement only for the whole workflow application. Two policies to automate advance reservation planning for individual tasks efficiently are presented and evaluated.

1 Introduction

With the emergence of more and more sophisticated services, Grid computing is becoming rapidly a popular way of providing support for many data intensive, scientific applications that, among other, may have large computational resource requirements. Such applications, without being embarrassingly parallel, may demonstrate a reasonably large degree of task parallelism. The specific paradigm we consider in this paper concerns Grid workflow applications. These applications require the execution of a list of tasks in a specific order. Most often, tasks and their dependences can be represented by a Directed Acyclic Graph (DAG). Several studies indicate that such DAG-like applications would constitute an important use case for emerging Grids [30, 4, 19].

DAG scheduling, as an optimization problem, has been well studied in the context of traditional homogeneous (and recently heterogeneous) parallel computing [12, 23, 29]. However, in the context of the Grid, the underlying environment is significantly different. Besides the heterogeneity and the possibly substantial communication overheads, there are issues related to the different administration domains that might be involved in providing resources for an application to run. All these may hinder the exploitation of parallelism. However, the most important characteristic of the environment is that the traditional model of running on homogeneous parallel machines, where a single local scheduler would be in charge, is no longer the norm. The consequence is that it cannot be guaranteed that the attempt to exploit parallelism may result in any performance improvements. For example, the parallel tasks may not actually execute in parallel on different resources (belonging to different administration domains) simply because of different behaviours that the job queue of each resource may adopt. In principle, this is due to the limited level of service that most current systems can offer; essentially this is summarized to "run a job whenever it gets to the head of the job queue". From the user's point of view, this might be perceived as lack of acceptable quality in the service offered when running onto a large, distributed, multi-site platform.

Advance Reservation of resources has been suggested as a means to guarantee that tasks will run onto a resource when the user expects them to run [17, 28]. Essentially, advance reservation specifies a precise time that jobs may start running. This allows the user to request resources from systems with different schedulers for a specific *time interval* (e.g., start time, finish time), thereby obtaining a sufficient number of resources for the time s(he) may need. Advance reservation has already received significant attention and has been considered an important requirement for future Grid resource management systems [25]. There has been already significant progress on supporting it by several projects and schedulers, such as the Load Sharing Facility platform (LSF) [16], Maui [10], COSY [6], and EASY [15, 27]; still, there is some scepticism in the community, especially with respect to the degree to which advance reservations contribute to improving the overall performance of a scheduler [9]. Various techniques have also been proposed to solve a number of problems stemming from advance reservation, such as reservation planning [31], Quality of Service [18] and resource utilization issues [13, 14, 21].

All existing work on advance reservation assumes that the environment consists of independent jobs competing for resources. However, in the context of workflow applications, such as those considered in [4, 19, 30], the workflow consists of a set of tasks linked by precedence constraints to a DAG. Although one might consider the whole workflow as a single job for which resources are negotiated and reserved for its whole duration (that is, start of the entry task until the finish of the exit task), this may lead to a waste of resources and low utilization: this is because precedence constraints and a varying degree of parallelism may leave resources without work to do. In that case, one may want to reserve resources for specific tasks. However, the reservation of tasks cannot be done without taking into account all other tasks in the DAG and, in particular, precedence constraints as well as the time that each task may need in order to complete (clearly, a child node in the DAG cannot start execution when a parent node is still running).

This paper focuses on the problem of planning advance reservations for the individual tasks of a DAG on a heterogeneous platform taking into account a user constraint in terms of the latest possible time that the execution of the whole DAG will be completed. In other words, we assume that the user specifies a time interval for which resources for the whole DAG are required. This time interval is determined by the time that the application can start running and the latest possible time that it can finish. Given this time interval, the problem relates to how to reserve appropriate time intervals for each task taking also into account the overall user constraint about the latest possible time that the whole application can finish.

The paper describes and evaluates two different strategies to solve the problem of finding individual task reservations. These strategies attempt to include sufficient 'extra time' to individual task reservations based on a user's request for the latest time that the whole execution of the DAG must finish. To the best of our knowledge, there has not been any prior work on this problem. The increasing interest in workflows in the context of the Grid requires studies to be undertaken at the level of finding appropriate strategies for planning reservations.

The remainder of the paper is organized as follows. Section 2 provides some background for the model used and the problem considered. Section 3 proposes two novel heuristics for task reservation in DAGs. Six different variants of the two heuristics have been implemented and are evaluated in Section 4. Finally, Section 5 concludes the paper.

2 Background

The model we use to represent the application, that is the DAG, and its associated information (e.g., estimated execution time of tasks and communication costs) is based on a model widely used in other heterogeneous computing scheduling studies [23, 29, 33]. A DAG consists of nodes and edges, where nodes (or tasks) represent computation and edges represent precedence constraints between nodes. The DAG has a single entry node and a single exit node. There is also a set of machines (resources) on which nodes can execute (usually, the execution time is different on each machine) and which need different time to transmit data. A machine can execute only one task at a time, and a task cannot start execution until all data from its parent nodes is available. An estimate for the execution time of each task on each machine is supposed to be known. Same, the amount of data that needs to be communicated between tasks is also known; along with an estimate for the communication cost between different machines, the last two values give the estimated data communication cost between two tasks that have a direct precedence constraint (that is, they are linked with an edge in the DAG) and they are running on specific (different) resources.

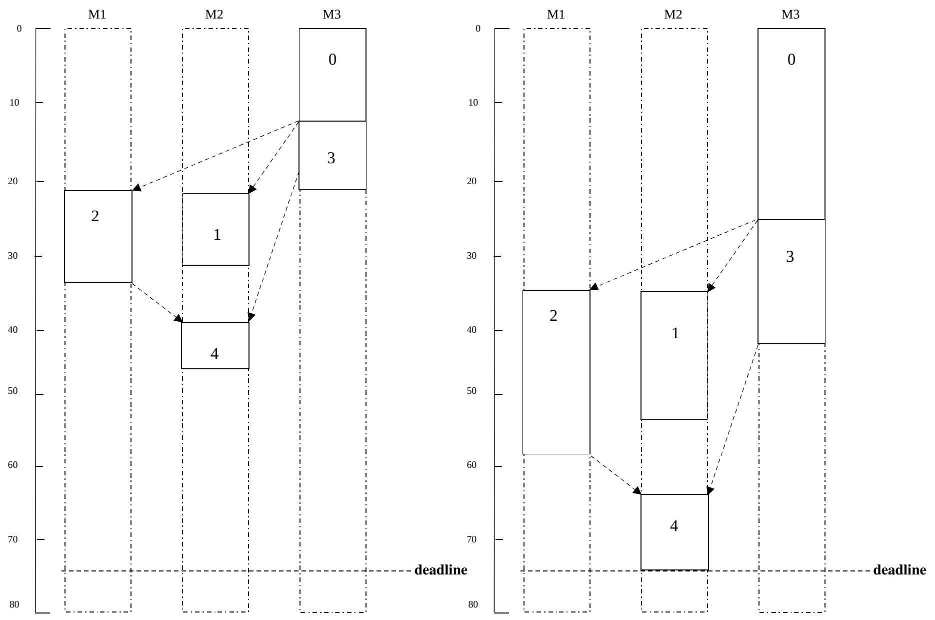
A number of papers have addressed the problem of minimizing the makespan when mapping the nodes of the DAG onto a set of heterogeneous machines; several algorithms, such as HEFT [29] or HBMCT [23], are known to provide good performance. It might be observed here that those algorithms could be used to provide an initial solution to the problem of planning advance reservations. In particular, these algorithms can provide a mapping of the tasks onto space and time (meaning on what machine a task will execute and what its starting time would be). As long as the overall makespan is smaller than the latest acceptable finish time for the whole application, one could plan reservations on the basis of this mapping.

However, there is one more subtle point to be made. The algorithms above provide a mapping on the basis of the estimated execution time of each task. In practice, the execution time of a task may differ significantly from the static estimate. Using advance reservation, if a job exceeds the time for which a resource has been reserved, it will, most likely, be killed (if re-negotiation is not possible). In the case of a DAG, killing one task would imply that all children tasks cannot start at their specified point in time (that is, the reservation slot for the resource); this may lead to an application failure, or, at best, the need to renegotiate the reservation of resources for the current task and all its descendants. If, for a moment, we consider advance reservation in the context of a single job rather than a DAG, it should be noted that, when making advance reservations, users are expected to reserve resources for a somewhat longer period of time than the time they predict their application will need. Certainly, performance prediction can never be perfect, however, adding some 'extra spare time' or 'slack' to the reservation will minimize the chances of their job getting killed (because it is still running at the end of the reservation slot). It would be against the whole concept of orchestrating and enacting workflows to expect that users would do the same at the task level with their workflows; instead, it is anticipated that users would specify requirements (and hence add some 'slack') for the whole workflow.

In previous work, it has been observed that, after scheduling a DAG, individual tasks in a DAG might include some 'slack' anyway, as a result of precedence and resource constraints (for example, think of the parent of a task, which finishes much earlier than all other parents of the task). In [24], the notion of *spare time* is introduced to represent the maximal delay that a task can afford to delay without affecting the start time of any of its dependent tasks (both on the DAG or on the same machine). Using this notion, assume a DAG, where the user has specified the latest acceptable finish time (or deadline) for the whole DAG, and an initial schedule has been constructed, using any conventional DAG scheduling algorithm, such as HEFT [29] or HBMCT [23]. Then, the problem becomes how to distribute fairly any extra time left between the finish time of the last task in the DAG and the latest acceptable finish time of the whole DAG, to the individual reservations of each task of the DAG, in such a way that each task gets the maximum possible amount of *spare time* comparing to the time it is predicted it will need [1]. Such a distribution would increase the *spare time* of each task (the spare time defined as above); it can be assumed safely that this would minimize the chances of an application failure due to the task still running at the end of its reservation slot.

To illustrate the above, consider the example schedule in Figure 1(a), where a simple DAG with 4 tasks has been mapped onto 3 machines. The problem is how to distribute to individual tasks the overall application spare time (that is, the user specified deadline for the overall application minus the finish time of

[1] Clearly, the assumption is that the maximum acceptable finish time for the whole application is greater than the finish time of the last task of the DAG as obtained by the initial schedule



(a) The Original Schedule (b) A Reservation Plan

Fig. 1. A Motivation Example.

Task 4). A possible distribution that provides to each task an amount of spare time approximately equal to its original execution time estimate is shown in Figure 1(b).

It should be mentioned that in several cases, the initial allocation of the tasks may give some spare time to some tasks as a result of parent-children relationships (for example, where one parent with a single child task finishes much earlier than the other parent) [24].

3 Towards a Solution of the Problem

3.1 Input and Notation

The input and the notation used is as follows:

– A workflow application is given; this is represented by a directed acyclic graph $G = (V, E)$, where V is a set of n tasks, and E is the set of edges representing flow of data between tasks.

```
(1) Phase 1: Obtain initial assignment by allocating each task in the
  given workflow (DAG) to a resource using a DAG scheduling algorithm.
(2) Phase 2:
  Repeat
    Compute the Application Spare Time
    Obtain a new allocation by selecting a policy for allocating
    this Application Spare Time to each task
  Until the Application Spare Time is zero or reaches a pre-defined value.
  The last allocation provides the final reservation plan.
```

Fig. 2. Advance Reservation Planning for DAG applications

- A set of (heterogeneous) resources is given. We assume that this set of resources qualifies to run all tasks of the DAG.
- For each task of the DAG, an estimated execution time on each machine is known. In addition, the amount of data that needs to be communicated between tasks is known, as well as the communication cost per data unit between different machines.
- An algorithm, alg, can be used to schedule the DAG onto the set of heterogeneous resources. This algorithm produces an initial mapping (or allocation) of tasks onto machines. This allocation is denoted by $alct$; the finish time of this allocation is $FinishTime_{alct}$. As noticed in the motivating example in the previous section, the initial allocation can be used to specify a reservation slot for each task (for example, see the slots for each task in Figure 1.a).
- A user specified maximum acceptable time by which the whole application (DAG) must finish is given by the user; this is denoted by $Deadline_G$. Note, that in real practice, users are expected to specify an earliest possible start time as well as a latest acceptable finish time. Without loss of generality, we consider the earliest possible start time to be equivalent to time zero in our setting.
- Finally, we define *Application Spare Time (AST)* to be the difference between $Deadline_G$ and $FinishTime_{alct}$, that is, $AST_{alct} = Deadline_G - FinishTime_{alct}$.

The purpose of the paper is to come up with an efficient strategy that would distribute the AST_{alct} to individual tasks, thereby extending their reservation slots (in a way similar to what we did in Figure 1(b) for the original schedule in Figure 1(a)) and making them more resilient to unexpected delays in their execution. This would minimize the chances that the application will need to re-negotiate resources (or even fail), because the execution of a task exceeds the time for which the resource has been reserved.

3.2 Outline of the Solution

Our strategy to come up with reservations for each task of the DAG consists of two phases is shown in Figure 2. In the first phase, an initial allocation of a given DAG application is constructed. Given a set of (heterogeneous) resources, the initial allocation is obtained using any algorithm for scheduling DAGs onto those resources in a way that minimizes the makespan (such as, [23, 29]). This allocation is constructed by taking into account estimated execution times for the tasks and for the communication. The initial allocation provides a start time and a finish time for each task assigned to a particular resource. If the makespan of this initial schedule exceeds the user deadline, this allocation is rejected and the user can be informed that the DAG cannot be scheduled within the required time.[2] If the makespan is less than the user deadline, the next phase is invoked.

In the second phase, the problem becomes how to distribute the *application spare time* to individual tasks in a way that each task has a sufficient spare time of its own, and, ideally, the application finish time becomes equal to the deadline specified by the user. Two strategies are used for this purpose — they are explained below.

3.3 Recursive Spare Time Allocation

The key idea of the first strategy is to use a formula to compute an amount of spare time to be added to each task on the basis of the overall application spare time. After such an amount of extra spare time is added to each task, the reservation slot of each task is appropriately extended and a new overall application spare time (smaller than the original, because of the extended reservation slots) is computed. This procedure is applied repeatedly until the overall application spare time becomes smaller than a threshold. The strategy is illustrated in Figure 3.

Four different formulae have been used to compute the amount of spare time to be added to each task:

1. The application spare time is divided evenly amongst all the tasks (this is the approach used in the description of the strategy in the Figure 3).
2. The application spare time is divided amongst tasks in such a way that each task gets the same percentage of spare time as a proportion to its estimated execution time (equivalent to the initially estimated reservation slot).
3. The application spare time is divided amongst tasks in such a way that each task gets the same percentage of spare time as a proportion to its estimated execution time, but at the first iteration spare time is given only to the tasks in the critical path of the allocation.
4. The application spare time is divided amongst tasks in such a way that each task gets the same percentage of spare time as a proportion to its estimated execution time. As opposed to the number 2 approach above, this approach takes into account, each time, the spare time that a current task may exhibit as a result of successor tasks starting not immediately after the end of the current task.

[2] This case, however, is beyond the scope of this paper. As already mentioned, we assume that the deadline specified by the user is always greater than the makespan achieved by the DAG scheduling algorithm.

```
Input:
    An application (workflow) represented by a DAG $G$ with $n$ tasks
    A set of machines
    A user defined deadline for the execution of the DAG, $Deadline_G$
    An initial schedule, $S$, built using any DAG scheduling algorithm (e.g., HBMCT),
        making use of estimates for the task execution time and the communication
    The initial schedule is used to generate for each task, $i$, a $ReservationSlot(i)$,
        which contains $task\_start\_time$, $task\_finish\_time$, $machine\_id$

Algorithm:
$totalST = 0$
$AST = Deadline_G - FinishTime_S$
repeat
  for each task $i = 1$ to $n$ do
    compute the Spare Time for $i$, $SpareTime(i)$
  end for
  //compute an amount of spare time to add to each task
  //for example, allocating the same amount of spare time to each task, as below
  $task\_spare\_time = AST \ / \ n$
  for each task $i = 1$ to $n$ do
    $totalST$ += $task\_spare\_time$
    if($SpareTime(i) < totalST$)
      extend $ReservationSlot(i)$ by $(totalST - SpareTime(i))$
    end if
  end for
  update Schedule $S$ with the new (extended) reservation slots
     (for each $ReservationSlot(i)$, $task\_start\_time$ and $task\_finish\_time$ are shifted
      to a later time that depends on the extension of the $ReservationSlot$ of
      the parents)
  $AST = Deadline_G - FinishTime_S$
  //$threshold$ is the criterion to exit the loop
  //its value can be 5% of the deadline for example
until ($AST < threshold$)
```

Fig. 3. The Recursive Spare Time Allocation Approach

3.4 The Critical Path Based Allocation

The critical path based policy tries to distribute the application spare time to the tasks on the critical path first (since those tasks determine the finish time of the application), and then it tries to balance the spare time of tasks in the remaining execution paths. The critical path based approach is shown in Figure 4. Same as before, two different formulae are used to compute the amount of spare time to be added to each task on the critical path:

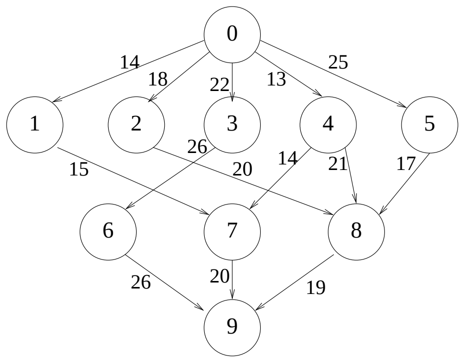
1. The application spare time is divided evenly amongst the tasks in the critical path (this is the approach used in the description of the strategy in the figure).

```
Input:
    An application (workflow) represented by a DAG $G$ with $n$ tasks
    A set of machines
    A user defined deadline for the execution of the DAG, $Deadline_G$
    An initial schedule, $S$, built using any DAG scheduling algorithm (e.g., HBMCT),
        making use of estimates for the task execution time and the communication
    The initial schedule is used to generate for each task, $i$, a $ReservationSlot(i)$,
        which contains $task\_start\_time$, $task\_finish\_time$, $machine\_id$

Algorithm:
$AST = Deadline_G - FinishTime_S$
for each task $i = 1$ to $n$ do
  $MinSpareTime(i) = AST$
end for
Find all paths $Paths$ in the initial schedule $S$ from the entry task in $G$ to the exit task
Find the critical path in the initial schedule $S$ and its tasks, $critical\_path\_tasks$
$num\_of\_cp\_tasks$ = number of $critical\_path\_tasks$
$cp\_spare\_time = AST \ / \ num\_of\_cp\_tasks$
for each task $i$ in $critical\_path\_tasks$
   $MinSpareTime(i) = cp\_spare\_time$
end for
for each other path $p$ in $Paths$ do // not the critical path
   $num\_of\_cp\_task\_this\_path$ = the number of critical path tasks on the path $p$
   $remain\_spare\_time = AST - num\_of\_cp\_task\_this\_path * cp\_spare\_time$
   $num\_of\_task\_this\_path$ = the number of tasks on $p$
   $remain\_tasks = num\_of\_task\_this\_path - num\_of\_cp\_task\_this\_path$
   $remain\_spare\_time\_each\_task = remain\_spare\_time / remain\_tasks$
   for each task $i$ in this path
      if ($MinSpareTime(i) > remain\_spare\_time\_each\_task$) then
         $MinSpareTime(i) = remain\_spare\_time\_each\_task$
      end if
   end for
end for
for each task $i = 1$ to $n$ do
      extend $ReservationSlot(i)$ by $MinSpareTime(i)$
end for
update Schedule $S$ with the new (extended) reservation slots
  (for each $ReservationSlot(i)$, $task\_start\_time$ and $task\_finish\_time$ are shifted to
   a later time that depends on the extension of the $ReservationSlot$ of the parents)
```

Fig. 4. The Critical Path Based Allocation Approach

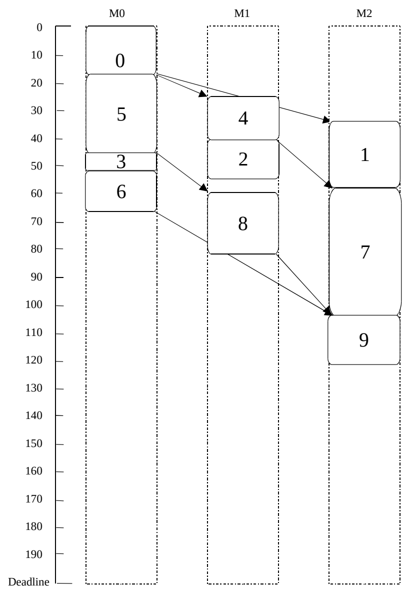
2. The application spare time is divided amongst tasks in the critical path in such a way that each task gets the same percentage of spare time as a proportion to its current execution time.



(a) an example graph

task	M0	M1	M2	task	M0	M1	M2
0	17	19	21	5	30	27	18
1	22	27	23	6	17	16	15
2	15	15	9	7	49	49	46
3	4	8	9	8	25	22	16
4	17	14	20	9	23	27	19

(b) the computation cost of nodes on three different machines



(d) the initial schedule derived by the HBMCT algorithm

processors	time for a data unit
m0 - m1	0.9
m1 - m2	1.0
m0 - m2	1.4

(c) the communication cost table for interconnected machines

	Reservation Slot			Reservation Slot	
task	start	finish	task	start	finish
0	0	17	5	17	47
1	36.6	59.6	6	51	68
2	42.7	57.7	7	59.6	105.6
3	47	51	8	62.3	84.3
4	28.7	42.7	9	105.6	124.6

(e) Reservation Slot of each task in the initial schedule on three different machines

Fig. 5. An example of reserving slots using a schedule generated by the HBMCT algorithm

3.5 An Example

An example with 10 tasks is used here to illustrate the two proposed approaches. The example workflow is shown in Figure 5(a); (b) gives the estimated computation cost of each task on 3 different machines, and (c) gives the commmunication costs between machines. Using the HBMCT DAG scheduling algorithm [23], the schedule is shown in Figure 5(d) with a makespan of 124.6; an initial reservation for each task of the workflow is built from this schedule with the starting time and finishing time of each task shown in Figure 5(e).

	task	spare time	allocated spare time	Slot (start)	Slot (finish)
Iteration 1 AST = 75.4	0	0	7.54	0	24.54
	1	0	7.54	44.14	74.68
	2	4.6	2.94	57.78	75.72
	3	0	7.54	62.08	73.62
	4	0	7.54	36.24	57.78
	5	0	7.54	24.54	62.08
	6	1.2	6.34	73.62	96.96
	7	0	7.54	74.68	128.22
	8	2.3	5.24	77.38	104.62
	9	0	7.54	133.36	159.9

Fig. 6. An example to illustrate the steps of the Recursive Spare Time Allocation Approach using the workflow in Figure 5.

path	num_cp_tasks	remaining ST	allocated spare time
0→1→7→9 (cp)	4	75.4	{(0, 18.85), (1, 18.85),(7, 18.85),(9, 18.85)}
0→5→3→6→9	2	37.7	{(0, 18.85), (5, 12.56),(3,12.56), (6, 12.56), (9, 18.85)}
0→4→2→8→9	2	37.7	{(0, 18.85), (4, 12.56),(2,12.56), (8, 12.56), (9, 18.85)}
0→4→7→9	3	18.85	{(0, 18.85), (4, 12.56), (6, 12.56), (9, 18.85)}
0→5→8→9	2	37.7	{(0, 18.85), (5, 12.56), (8, 12.56), (9,18.85)}

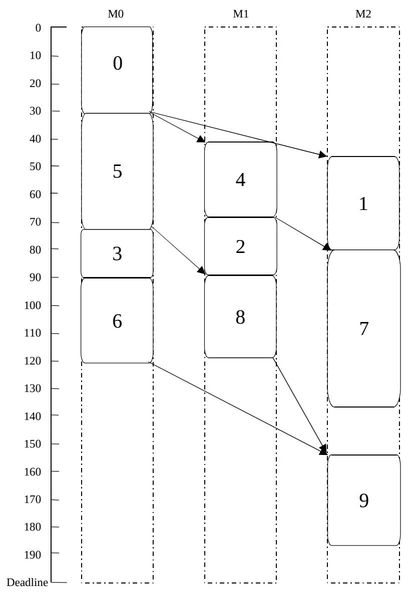
(a) reservation steps

task	Slot (start)	Slot (finish)	task	Slot (start)	Slot (finish)
0	0	35.85	5	35.85	78.41
1	55.45	97.3	6	94.97	124.53
2	74.11	101.67	7	97.3	162.15
3	78.41	94.97	8	101.67	136.23
4	47.55	74.11	9	162.15	200

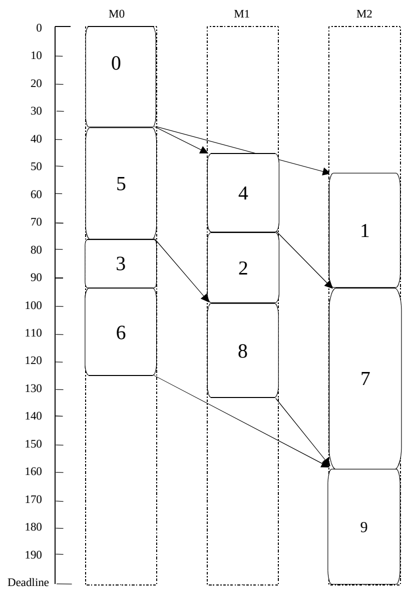
(b) The final reservation slot of each task

Fig. 7. An example to illustrate the steps of the Critical Path Based Allocation Approach using the workflow in Figure 5.

Assume a deadline of 200 for the whole workflow is required from the user. Then, using the schedule above, the initial Application Spare Time (AST) to be distributed to tasks is equal to $200 - 124.6 = 75.4$. Figure 6 shows the first iteration of the Recursive Spare Time Allocation approach. The approach computes the spare time of each task and allocates the same amount of spare time to each task apart from the ones already having some spare time. Those tasks will be allocated the difference only. For instance, task 2 had spare time of 4.6 from the initial schedule, therefore, another $2.94 (= 7.54 - 4.6)$ is allocated to it in the new reservation slot. After two more iterations, where additional spare time is added to each task, the reservation slots for each task and the final schedule are shown in Figure 8(a).



(a) reservation slots for each task using the Recursive Spare Time Allocation Approach (the threshold is 5%)



(b) reservation slots for each task using the Critical Path Based Allocation Approach

Fig. 8. The final reservation of each task of the workflow in Figure 5(a) using the proposed two approaches.

Figure 7 shows the reservation steps using the Critical Path Based Allocation approach. All paths in the initial schedule are found, and the tasks which are in the critical path (which is {0, 1, 7, 9}) obtain the same amount of time by dividing the AST evenly. The spare time for the remaining tasks is computed by dividing the remaining amount of AST in the path. Only the smallest amount of spare time that each task may obtain from different paths will count. For instance, the spare time of task 5 on the (scheduled) path {0, 5, 3, 6, 9} is 12.56, and on the other path {0, 5, 8, 9}, the amount for task 5 is 18.85; however, only the smallest amount, 12.56, counts to the final reservation slot. The reservation slots for each task and the final schedule are shown in Figure 8(b).

4 Experimental Results

4.1 The Setting

We evaluated the performance of the proposed strategies in terms of their ability to distribute the application spare time to the individual tasks as well as their behavior with respect to possible failures at run-time due to differences from the predicted task execution times. For the evaluation we used simulation.

Both strategies described above (and all their variants, that is, a total of six variants) are implemented. The six variants are denoted by r_even_time, $r\_even\_percent1$, r_cp_first, $r\_even\_percent2$, for the recursive spare time allocation strategy (in the order they were presented in Section 3.3), and cp_even_time, and $cp\_even\_percent$ for the critical path based strategy (again, in the order they were presented in Section 3.4).

Four different DAG scheduling algorithms have been used to obtain the initial allocation: Hybrid.BMCT [23], FCP [22], DLS [26] and HEFT [29].

Five different types of DAGs have been used for the evaluation. The first corresponds to a real-world workflow application, Montage [3, 4]. The second corresponds to generic Fork&Join DAGs; the structure can be seen an abstraction of Montage. It consists of repetitive layers where in each layer a number of tasks are spawned to be joined again in the next layer. The number of tasks that are spawned each time is decreased by 1. The third and fourth types of DAGs correspond to Fast-Fourier-Transform (FFT) and Laplace operations [11]; comparing to the previous DAGs their structure is fully symmetric. These two graphs have been extensively used in several studies related to DAG scheduling [2, 22–24]. Finally, the fifth type aims to provide a more unstructured type of DAGs and is randomly generated as follows. Each graph has a single entry and a single exit node; all other nodes are divided into levels, with each level having at least two nodes. Levels are created progressively; the numbers of nodes at each level is randomly selected up to half the number of the remaining (to be generated) nodes. Care is taken so that each node at a given level is connected to at least one node of the successor level and *vice versa*.

All five types of DAGs have been used by a plethora of studies related to DAG and workflow scheduling in the literature [4, 19, 2, 22, 24, 23, 30, 33]. In our experiments, we used DAGs of about 60 tasks each (this is approximately 60, because some types of DAG cannot generate DAGs of exactly 60 tasks). We always assumed that 5 machines were available. Regarding the estimated execution time of each task on each different machine: this is randomly generated from a uniform distribution in the interval [10,100], for the last 4 types and the interval [50,100] for Montage, while the communication-to-computation ratio (CCR) is randomly chosen from the interval [0.1, 1].

Two sets of experiments were carried out. The first set evaluates the performance of each variant in terms of the spare time assigned to each task. For the comparison, we assume a fixed deadline. However, given that each algorithm may generate a different schedule, the makespan of the initial allocation is expected to differ; this means that the application spare time to be distributed to tasks may be different (since the deadline is always the same) depending on the original DAG scheduling algorithm used. Thus, we present the application spare time as a percentage ratio of the corresponding makespan (i.e., the $FinishTime_{alct}$), as follows:

$$\alpha = (AST_{alct}/FinishTime_{alct}) \times 100.$$

In general, the smaller the value of α is, the tighter the required deadline would be, comparing to the makespan of the initial schedule; consequently, the less the spare time that can be distributed to each task (although, as a result of a seemingly inefficient schedule in terms of the overall makespan, tasks may get already a high spare time inherent in the schedule).

The second experiment considers run-time execution time deviations from the estimated execution time of each task (that was used to plan their reservations) and evaluates how well the strategies can accommodate those deviations.

Finally, we also evaluate the running time of each variant.

4.2 Performance Results

Distribution of Spare Time Using a common deadline in all cases (which we assume it is at time 1500 after the start of the first task of the DAG), the six variants are evaluated using four different DAG scheduling algorithms for the initial allocation and five different types of DAGs. In each case, we are interested to find out how well each variant distributes the application spare time to individual tasks. Thus, for all tasks of the DAG, we find the minimum, average, and maximum spare time for a task (denoted by Min, Avg, Max) as a percentage of the task's estimated execution time. The minimum spare time percentage is the most important indicator, since it shows the highest percentage of deviation from the estimated execution time of a task that can be afforded by any task without exceeding the reserved timeslot.

The results, averaged over 100 runs, are shown in Table 1. Several observations can be made:

- It appears that all six different variants manage to achieve a reasonable distribution of the spare time to each task as can be seen by observing the minimum spare time percentage (which is for each task analogous to what the value of α is for the whole DAG). In most cases, this seems to be close to or higher than the corresponding value of α. It also appears that the critical path based approaches (cp_even_time and cp_even_percent) lead to slightly higher values for the minimum spare time percentage. It is interesting to notice that, for the Montage workflow, HBMCT manages to guarantee a minimum spare time percentage of 47.8% for each task, even though the value of α is only 34.
- On the DAG scheduling algorithm front, it is interesting to notice that HBMCT generally shows the highest minimal spare time percentage (the only exception being FFT graphs, where the DLS algorithm performs better, by about 5%, in 4 out of the 6 variants). It is worth to notice also that, even though it has a lower α value (that is, a longer makespan) the FCP algorithm outperforms HEFT (which has a higher α value and hence more application spare time to distribute to individual tasks) in terms of the spare time percentage, for all types of DAG except Montage. This can be attributed to the inefficient initial schedule that FCP builds, which already gives a rather large amount of spare time to each task. Still, however, FCP is outperformed by HBMCT.

		HBMCT α=34			FCP α=7			DLS α=32			HEFT α=24		
Montage		*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*
	r_even_time	38.3	110.2	66.5	20.4	157.7	48.2	36.7	108.6	62.7	26.8	140.4	56.9
	r_even_percent1	40.8	107.5	69.8	22.3	146.8	45.8	34.1	108.0	68.4	27.5	132.0	58.3
	r_cp_first	40.0	102.9	70.2	23.1	135.8	44.4	35.4	111.6	65.9	32.7	134.5	59.9
	r_even_percent2	42.6	108.3	72.5	22.7	149.5	46.8	35.4	115.7	67.9	28.4	131.7	60.5
	cp_even_time	43.9	111.6	78.7	24.4	159.6	50.5	35.7	122.7	70.2	30.7	148.2	62.5
	cp_even_percent	47.8	98.7	78.8	23.9	160.6	55.1	40.1	115.9	68.4	34.9	145.1	61.6
		HBMCT α=24			FCP α=9			DLS α=23			HEFT α=16		
Random		*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*
	r_even_time	21.2	200.6	55.3	18.3	186.4	47.6	18.2	193.2	46.9	15.6	172.7	42.6
	r_even_percent1	25.9	178.6	60.4	19.9	179.0	48.2	17.5	187.9	44.1	15.0	160.6	41.1
	r_cp_first	24.0	203.8	55.4	19.2	180.9	46.8	19.6	192.6	46.6	15.9	172.2	44.0
	r_even_percent2	23.6	168.4	56.6	19.2	178.3	48.9	17.5	182.6	42.8	15.2	154.5	35.6
	cp_even_time	27.6	194.7	58.3	20.8	177.1	48.7	20.2	184.4	48.4	16.5	169.9	37.2
	cp_even_percent	25.6	172.2	55.3	19.6	178.2	49.9	22.0	170.4	50.7	16.6	155.4	40.1
		HBMCT α=28			FCP α=11			DLS α=28			HEFT α=18		
Laplace		*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*
	r_even_time	25.4	205.7	62.3	23.0	188.1	55.5	24.3	208.4	66.2	18.7	175.4	56.9
	r_even_percent1	23.3	187.7	70.4	21.6	166.0	62.4	21.7	182.1	64.9	18.7	168.4	57.1
	r_cp_first	26.8	215.5	70.5	23.1	182.4	56.9	23.7	198.5	66.5	20.4	177.5	54.8
	r_even_percent2	26.0	185.7	72.3	19.2	173.8	59.3	25.5	190.8	73.8	20.0	172.5	60.5
	cp_even_time	27.1	206.4	66.9	23.0	178.2	55.5	24.1	197.4	67.4	23.8	189.4	56.9
	cp_even_percent	28.4	196.9	68.6	24.6	171.7	56.6	25.9	192.7	67.3	21.2	172.6	58.8
		HBMCT α=19			FCP α=7			DLS α=18			HEFT α=12		
F&J		*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*
	r_even_time	19.3	165.2	50.5	17.5	156.6	52.7	18.7	151.7	52.9	15.2	140.9	50.5
	r_even_percent1	18.5	156.8	51.6	17.3	154.3	54.9	19.7	139.5	55.5	14.9	142.4	48.4
	r_cp_first	20.2	161.8	55.8	19.5	163.9	56.8	21.0	158.7	55.0	18.4	151.6	50.6
	r_even_percent2	18.9	150.5	53.0	18.2	159.8	59.3	20.3	145.1	57.9	17.0	151.4	50.7
	cp_even_time	19.6	160.3	57.7	17.7	164.8	60.6	19.1	167.6	58.9	17.8	151.9	54.8
	cp_even_percent	19.5	155.4	60.3	18.8	151.7	60.1	20.7	160.6	70.2	17.6	146.3	57.3
		HBMCT α=21			FCP α=8			DLS α=21			HEFT α=15		
FFT		*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*
	r_even_time	21.6	198.7	56.7	17.9	170.6	54.0	22.3	193.7	57.5	18.2	180.6	55.7
	r_even_percent1	23.6	185.3	58.3	16.7	165.6	57.4	22.2	176.6	60.9	20.2	164.9	58.6
	r_cp_first	20.7	192.9	60.6	18.2	171.8	57.0	23.1	203.0	62.5	21.6	200.4	60.8
	r_even_percent2	24.1	182.4	60.4	20.2	159.8	58.2	23.7	179.0	61.9	23.6	196.9	60.5
	cp_even_time	23.2	191.9	60.6	21.9	181.6	59.1	25.7	190.4	62.4	24.6	199.8	62.0
	cp_even_percent	24.6	187.1	61.8	20.4	176.6	60.2	24.9	195.1	65.8	23.9	191.7	62.5

Table 1. Minimum, maximum, and average spare time as a percentage of the estimated execution time of each task using: 6 approaches to distribute spare time to tasks; 4 different DAG scheduling algorithms to obtain the initial schedule; 5 different types of DAGs of about 60 tasks on average; and scheduling on 5 machines. In all cases, the user specified deadline is 1500.

– The different types of DAGs, although they generate different results, still exhibit a consistent behaviour. The only exception arises for the Montage workflow and in relation to the FCP algorithm. It can be speculated that, although an originally inefficient schedule (as the one produced by the FCP algorithm) may have some inherent spare time, this is not necessarily fairly distributed amongst tasks. There might be an argument here in favour of algorithms where a carefully produced original schedule (not necessarily optimized for minimum makespan) already includes some spare time carefully distributed among tasks, but this remains to be investigated.

Evaluation of the behaviour of our approach with run-time changes The second set of experiments examines how well the proposed approaches behave in a realistic environment, where they need to accommodate deviations from the estimated execution time of each task at run-time. In order to emulate run-time changes (in relation to the estimated execution times) we adopt the notion of *Quality of Information* (QoI) [24]. This represents an upper bound on the percentage of error that the statically estimated execution time may have with respect to the actual execution time. So, for example, a percentage error of 10% would indicate that the actual run-time execution time of a task will be within 10% (plus or minus) of the static estimate for the task. Clearly, in this case, if the planned reservations for each task have a spare time higher than 10% (as a percentage of the task's estimated execution time), the actual execution time of a task cannot exceed its reservation slot.

In this set of experiments, we used only the HBMCT algorithm, since it appears to perform generally better than the other algorithms considered earlier. We also consider only the Montage workflow. We consider different values for α (20, 50, 100, 150) and QoI (equal to 20%, 50%, 100%, 150% of the estimated execution time). Our aim is to evaluate the number of failures (a failure means that one task of the DAG could not complete its execution within its reserved slot) as well as the utilization of the reserved slots (this is the average utilization of the reserved slots for each machine). For comparison purposes, the six variants proposed in this paper are compared against an approach which reserves all resources that might be needed for the entire execution of the DAG (*DAG_Reserve*). The results are shown in Table 2. Same as before, the experiment is repeated 100 times and 5 machines are considered.

The main observation is that, generally, if the value of the QoI is less than the value of α it is unlikely to have failures (the only exception seems to arise for the largest value of QoI, 150%). This can be justified using the results in the previous set of experiments, where it was observed that, generally, the minimum spare time percentage that can be added to each task is close to the value of α. This means that for deviations in the task execution time that are up to about α%, the reservation plan is quite resilient and no (or very few) failures are expected. The main lesson from this observation is that all that users need to do when asking for resources for a workflow is to specify the amount of 'slack' that they would be prepared to afford for the execution of their workflow: this should be roughly related to the maximum deviation that they expect from the estimated execution time of each task in the workflow. Individual reservation slots for each task can then be derived automatically using appropriate heuristics.

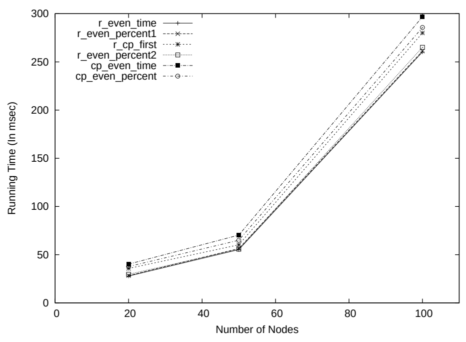
Comparing the variants proposed in this paper with the approach that reserves all resources throughout the entire DAG's execution, it can be seen that the former is more robust to failures (not surprising, given that in our variants the spare time of the whole DAG is distributed to individual tasks) but it suffers from low utilization within the reserved slots. It should be noted here that these values do not take into account the fact that our variants, which are based on individual task reservations, leave 'gaps' in the resources while the DAG is being

		Number of Failures				Reserved Slot Utilization				
α	QoI	*20%*	*50%*	*100%*	*150%*	*0%*	*20%*	*50%*	*100%*	*150%*
20	*r_even_time*	0	22	85	100	60.1	66.2	75.7	82.8	92.5
	r_even_percent1	0	20	84	100	59.4	65.6	74.3	81.3	90.3
	r_cp_first	0	22	84	100	59.3	66.8	76.0	81.9	92.0
	r_even_percent2	0	21	80	100	58.7	65.4	76.1	81.5	91.1
	cp_even_time	0	21	83	100	60.4	67.0	77.1	82.4	91.7
	cp_even_percent	0	21	82	100	60.6	66.6	75.9	82.1	91.9
	DAG_Reserve	0	8	59	100	44.7	46.4	49.5	53.5	58.8
50	*r_even_time*	0	0	19	46	52.5	59.4	70.2	82.6	91.9
	r_even_percent1	0	0	17	45	51.4	58.0	68.9	80.4	90.1
	r_cp_first	0	0	17	44	52.7	58.5	68.7	80.8	90.2
	r_even_percent2	0	0	17	44	52.4	58.7	69.3	81.0	90.4
	cp_even_time	0	0	16	44	52.1	59.0	69.5	80.9	89.8
	cp_even_percent	0	0	16	45	51.9	58.7	69.1	80.4	89.4
	DAG_Reserve	0	0	7	29	35.4	36.7	38.7	40.4	43.8
100	*r_even_time*	0	0	0	13	27.4	33.1	40.4	64.4	78.0
	r_even_percent1	0	0	0	10	26.3	31.4	38.8	63.1	77.9
	r_cp_first	0	0	0	11	25.9	32.0	38.7	63.2	77.0
	r_even_percent2	0	0	0	10	26.4	32.7	39.0	63.7	76.6
	cp_even_time	0	0	0	11	26.4	31.8	39.2	63.0	76.9
	cp_even_percent	0	0	0	11	26.2	31.6	39.0	63.2	77.3
	DAG_Reserve	0	0	0	5	20.3	21.5	25.0	27.9	31.3
150	*r_even_time*	0	0	0	7	21.7	25.8	35.5	46.4	61.2
	r_even_percent1	0	0	0	4	21.5	24.6	34.8	45.5	59.6
	r_cp_first	0	0	0	5	21.2	24.8	34.6	46.0	59.8
	r_even_percent2	0	0	0	5	20.9	24.7	34.0	46.0	60.5
	cp_even_time	0	0	0	5	21.2	24.4	34.0	46.4	60.7
	cp_even_percent	0	0	0	5	21.2	24.4	33.6	46.1	60.9
	DAG_Reserve	0	0	0	0	14.9	16.7	19.6	22.6	26.8

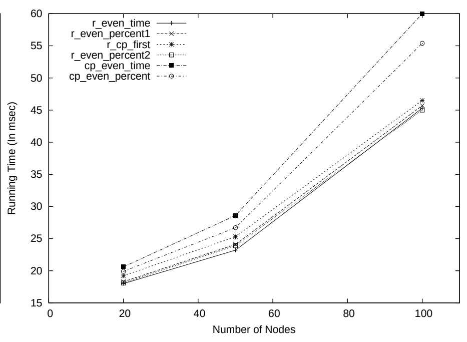
Table 2. Number of failures (reservation slot exceeded) and average reserved slot utilization for each of 6 task reservation approaches and DAG reservation approach with different QoI and α values. Results obtained over 100 runs using Montage workflows each with 57 tasks and scheduling on 5 machines with HBMCT algorithm.

executed. Thus, our variants allow to regain unused resource time after a job has been completed by *backfilling* [21] other, independent jobs that do not have advance reservation. This creates a better potential to increase overall resource utilization. Instead, in the case where the resources are reserved for the entire DAG, backfilling would not be desirable until the exit task of the DAG has been completed.

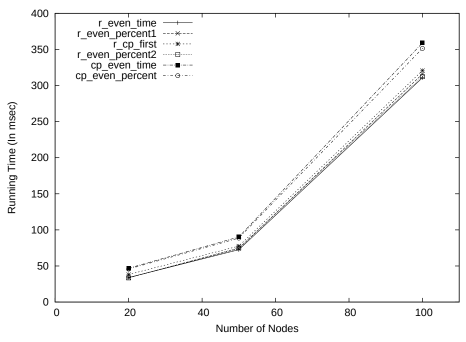
Running Time Although the two strategies that were compared in the previous section perform similarly, the variants based on the recursive based strategy achieve the same result at a significantly reduced cost. Figure 9 shows how the



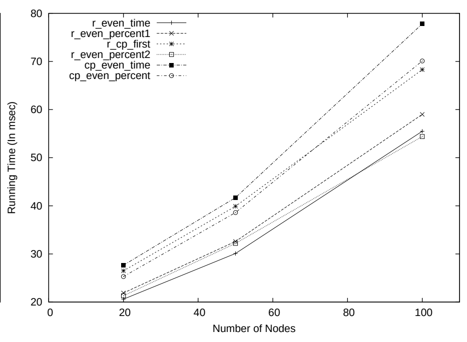
(a) HBMCT



(b) FCP



(c) DLS



(d) HEFT

Fig. 9. Average running time (over 100 runs on randomly generated DAGs) of six different reservation planning variants and four different DAG scheduling algorithms.

running time varies for each variant considered. The experiment was carried out with random DAGs (since they provide us with more flexibility in specifying a different number of tasks for the DAG) having 20 to 100 nodes each; the reservation plans considered an alpha value of 50. It can been seen that the critical path based policies lead to faster increases in the running time than the recursive based ones as the number of nodes in the DAG increases. This is because finding every path from the entry node to the exit node in the allocated schedule takes a significant amount of time. This may indicate that the critical path based variants, although they have the potential to perform slightly better, they come with an extra cost.

5 Conclusion

This paper presented two novel advance reservation policies for workflows, which attempt to distribute the spare time between an initial schedule (obtained by any DAG scheduling algorithm) and the deadline for the execution of the workflow gracefully to each task, in order to cope with run time execution time changes for each task. The approaches are based on either recursively allocating the time to each task or optimizing the critical path tasks. The strategies were designed to be usable by any DAG scheduling algorithm.

The main outcome of this work has been the proposal of efficient heuristics that can automate the process of coming up with reservation slots for scheduling individual tasks of a workflow (DAG), in the context of a system allowing advance reservations, without user intervention. In line with the philosophy for workflow automation in current research, all that the user needs to specify is the latest acceptable finish time for the whole workflow. As illustrated in the paper, the rest can be automated using a combination of appropriate heuristics.

Further evaluation could consider the heuristics presented in this paper in conjunction with a more dynamic environment, where DAGs as well as other jobs, not necessarily having advance reservations, co-exist. Such an environment could allow more complete analysis of resource utilization and performance by applying backfilling and/or techniques for dynamic re-planning advanced reservations based on run-time information.

Acknowledgements

We are grateful to the anonymous referees whose comments helped improve the paper. The work carried out in this paper has been funded by EPSRC, UK (grant reference GR/S67654/01) whose support we are pleased to acknowledge.

References

1. A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra. A Unified Resource Scheduling Framework for Heterogeneous Computing Environments. In the *8th Heterogeneous Computing Workshop (HCW)*, 1999.
2. O. Beaumont, V. Boudet, and Y. Robert. The Iso-Level Scheduling Heuristic for Heterogeneous Processors. *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP2002)*, 2002 (extended version available as Research Report RR2001-22, LIP, ENS Lyon, France).
3. G. B. Berriman, J. C. Good, A. C. Laity, A. Bergou, J. Jacob, D. S. Katz, E. Deelman, C. Kesselman, G. Singh, M. Su and R. Williams. Montage: A Grid Enabled Image Mosaic Service for the National Virtual Observatory. In the Conference Series of *Astronomical Data Analysis Software and Systems XIII (ADASS XIII)*, Volume 314, 2004.
4. J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Resource Allocation Strategies for Workflows in Grids. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005).*
5. L. Boloni, and D. C. Marinescu. Robust scheduling of metaprograms. In *Journal of Scheduling*, 5:395-412, 2002.
6. J. Cao and F. Zimmermann. Queue Scheduling and Advance Reservation with COSY. In the *Proceedings of 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, USA, April 2004.

7. H. Casanova, A. Legrand, D. Zagorodnov and F. Berman. Heuristics for scheduling parameter sweep applications in Grid environments. In *9th Heterogeneous Computing Workshop (HCW'00)*, 2000.
8. I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. *Proceedings of the International Workshop on Quality of Service*, 1999.
9. D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel Job Scheduling — A Status Report. In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.
10. D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2221, pp. 87-102, 2001.
11. Y.-K. Kwok and I. Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7(5):506-521, 1996.
12. Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs. *ACM Computing Surveys*, 31(4):406-471, 1999.
13. B. Lawson and E. Smirni. Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems. In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, Lect. Notes Comput. Sci. Vol. 2537, pp. 72-87, 2002.
14. B. Lawson, E. Smirni, and D. Puiu. Self-adapting Backfilling Scheduling for Parallel Systems. In the *Proceedings of the 2002 International Conference on Parallel Processing (ICPP 2002)*, pages 583-592, Vancouver, B.C., August 2002.
15. D. A. Lifka, M. W. Henderson, and K. Rayl. Users Guide to the Argonne SP Scheduling System. Technique Report ANL/MCS-TM-201, Argonne National Laboratory.
16. Load Sharing Facility platform. *http://www.platform.com/products/LSF/.*
17. J. MacLaren. Advance Reservations: State of the Art. In Global Grid Forum 9 (GGF9), Scheduling and Resource Management Workshop, Chicago, USA, October 2003.
18. J. MacLaren, R. Sakellariou, K. T. Krishnakumar, J. Garibaldi, and D. Ouelhadj. Towards Service Level Agreement Based Scheduling on the Grid. In *Workshop on Planning and Scheduling for Web and Grid Services* (in conjunction with ICAPS-04), June 3-7, 2004, pp. 100-102.
19. A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu and L. Johnsson. Scheduling Strategies for Mapping Application Workflows onto the Grid. In *IEEE International Symposium on High Performance Distributed Computing (HPDC 2005)*, 2005.
20. R. Min and M. Maheswaran. Scheduling Co-Reservations with Priorities in Grid Computing Systems. *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2002.
21. A. W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *IEEE Transactions on Parallel and Distributed Computing*, volume 12, pages 529-543, 2001.
22. A. Radulescu and A.J.C. van Gemund. Low-Cost Task Scheduling for Distributed-Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6), pp. 648-658, June 2002.

23. R. Sakellariou and H. Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proceedings of 13th Heterogeneous Computing Workshop (HCW 2004)*, 26-30 April 2004, Santa Fe, New Mexico, USA.
24. R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4), Dec. 2004.
25. U. Schwiegelshohn, P. Wieder and R. Yahyapour. Resource Management for Future Generation Grids. In *Future Generation Grids*, V. Getov, D. Laforenza, A. Reinefeld (Eds.), Springer, CoreGrid Series, 2005.
26. G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, February 1993.
27. J. Skonira, W. Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler API Project. In the *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, Springer-Verlag Lecture Notes In Computer Science, Vol. 1162, pp. 41-47, 1996.
28. W. Smith, I. Foster, and V. Taylor. Scheduling with Advanced Reservations. In the *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, pages 127-132, May 2000.
29. H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.
30. M. Wieczorek, R. Prodan and T. Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. In *SIGMOD Record*, volume 34(3), September 2005.
31. V. Yarmolenko and R. Sakellariou. An Evaluation of Heuristics for SLA Based Parallel Job Scheduling. In *3rd High Performance Grid Computing Workshop* (in conjunction with IPDPS 2006), IEEE Computer Society, 2006.
32. H. Zhao and R. Sakellariou. A Low-Cost Rescheduling Policy for Dependent Tasks on Grid Computing Systems. In the *Proceedings of the 2nd AcrossGrids*, January 28-30, 2004. Lecture Notes in Computer Science, Vol. 3165, page 21-31, Springer.
33. H. Zhao and R. Sakellariou. An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In the *Proceedings of 9th International Euro-Par Conference*, LNCS 2790, Springer, 2003.