Scientific Programming 12 (2004) 253–262
IOS Press

A low-cost rescheduling policy for efficient mapping of workflows on grid systems

Rizos Sakellariou and Henan Zhao
School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK
E-mail: {rizos, hzhao}@cs.man.ac.uk

Abstract. Workflow management is emerging as an important service in Grid computing. A simple model that can be used for the representation of certain workflows is a directed acyclic graph. Although many heuristics have been proposed to schedule such graphs on heterogeneous environments, most of them assume accurate prediction of computation and communication costs. This limits their direct applicability to a dynamically changing environment, such as the Grid. In this environment, an initial schedule may be built based on estimates, but run-time rescheduling may be needed to improve application performance. This paper presents a low-cost rescheduling policy, which considers rescheduling at a few, carefully selected points during the execution. This policy achieves performance results, which are comparable with those achieved by a policy that dynamically attempts to reschedule before the execution of every task.

1. Introduction

Many use cases of Grid computing relate to applications that require complex *workflows* to be mapped onto a range of distributed resources. Although the characteristics of workflows may vary, a simple approach to model a workflow is by means of a *directed acyclic graph* (DAG) [8,10]. This model provides an easy way of addressing the mapping problem; a schedule is built by assigning the nodes (the terms 'task' and 'node' are used interchangeably throughout this paper) of the graph onto resources in a way that respects task dependences and minimizes the overall execution time. In the general context of heterogeneous distributed computing, a number of scheduling heuristics have been proposed (see [15,17,19] for an extensive list of references). Typically, these heuristics assume that accurate prediction is available for both the computation and the communication costs. However, in a real environment and even more in the Grid, it is difficult to estimate accurately those values due to the dynamic characteristics of the environment. Consequently, an initial schedule may be built using inaccurate predictions; even though the schedule may be optimized with respect to these predictions, run-time variations may affect the schedule's performance significantly.

There are two main approaches to deal with unpredictability. One approach is to schedule all tasks at run-time, as they become available; this may take place on a per task basis or in groups of independent tasks (as in [7]). The other approach is to plan in advance, build a static schedule using the available estimates, and possibly respond to changes that may occur at run-time by *rescheduling*. In the context of the Grid, rescheduling of one kind or the other has been considered by a number of projects, such as AppLeS [2,6], Condor-G [9], Data Grid [11] and Nimrod-G [4,5]. However, all these projects consider the dynamic scheduling of sets of independent tasks. For DAG rescheduling, a hybrid remapper based on list scheduling algorithms was proposed in [14]. Taking a static schedule as the input, the hybrid remapper uses the run-time information that obtained from the execution of precedence nodes to make a prediction for subsequent nodes that is used for remapping.

Generally speaking, rescheduling adds an extra overhead to the scheduling and execution process. This may be related to the cost of reevaluating the schedule as well as the cost of transferring tasks across machines (in this paper, we do not consider pre-emptive policies at the task execution level). This cost may be offset by gains in the execution of the schedule; however,

ISSN 1058-9244/04/$17.00 © 2004 – IOS Press and the authors. All rights reserved

what appears to give an indication of a gain at a certain stage in the execution of a schedule (which may trigger a rescheduling), may not turn to be good later in the schedule. In this paper, we attempt to strike a balance between the cost of rescheduling and the performance of the schedule. We propose a novel, *low-cost*, rescheduling policy, which improves the initial static schedule of a DAG, by considering *only* selective tasks for rescheduling based on measurable properties; as a result, we call this policy *Selective Rescheduling* (SR). Based on simulation results (the results presented here complement and expand the results included in the conference version of this paper [21]), this policy gives equally good performance with policies that consider rescheduling for every task of the DAG, at a much lower cost. In our experiments, SR considers less than 30% of the tasks of the DAG for rescheduling; in most cases, this number is even less than 20%.

The remainder of this paper is organized as follows. Section 2 defines two criteria to represent the robustness of a schedule, spare time and the slack. We use these two criteria to make decisions for the *Selective Rescheduling* policy, presented in Section 3. Section 4 evaluates the performance of the policy. Finally, Section 5 concludes the paper.

2. Preliminaries

The model used in this paper to represent an application is the *directed acyclic graph* (DAG), where nodes (or tasks) represent computation and edges represent communication (data flow) between nodes. The DAG has a single entry node and a single exit node. There is also a set of machines on which nodes can execute (with a different execution cost 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. The scheduling problem is to assign the tasks onto machines so that precedence constraints are respected and the makespan (i.e., the length of the schedule) is minimized. A solution to this problem is found using an appropriately designed heuristic [15,17,19]; the solution, called *schedule*, can be regarded as a quadruplet, which, for each task, specifies the machine on which it has been scheduled for execution, as well as, start time and finish time. For an example, see Fig. 1.

Previous work has attempted to characterize the robustness of a schedule; in other words, how robust the schedule would be if variations in the estimates used to build the schedule were to occur at run-time [1,3]. Although the robustness metric might be useful in evaluating overall different schedules, it has little direct value for our purposes; here, we wish to use specific criteria to select, at run-time, particular tasks before the execution of which it would be beneficial to reschedule. To achieve this, we build on and extend two fundamental quantities that have been used to measure robustness; the *spare time*, and the *slack* of a node. The spare time, computed between a pair of dependent nodes that are either connected by an edge in the DAG (data dependence), or are to be executed successively on the same machine (machine dependence), shows what is the maximal time that the source of dependence can execute *without* affecting the start time of the sink of the dependence. The slack of a node is defined as the minimum spare time on any path from this node to the exit node of the DAG. This is the maximum delay that can be tolerated in the execution time of the node without affecting the overall schedule length. If the slack of a node is zero, the node is called *critical*; any delay on the execution time of this node will affect the makespan of the application.

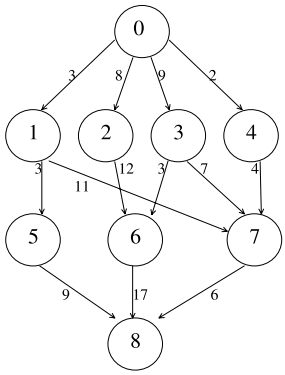
A formal definition and an example follow below. We note that the definitions in [3] do not take into account the communication cost between data dependent tasks, thereby limiting their applicability. Our definitions are augmented to take into account communication.

2.1. Spare time

Consider a schedule for a given DAG; the spare time between a node i and an immediate successor j is defined as

$$\mathrm{Spare}_{\mathrm{DAG}}(i,j) = ST(j) - DAT(i,j),$$

where $ST(j)$ is the expected start time of node j (on the machine where it has been scheduled to), and $DAT(i,j)$ is the time that all the data required by node j from node i will arrive on the machine where node j executes. To illustrate this with an example, consider Fig. 1 and the schedule in Fig. 1(d) (derived using the HEFT heuristic [19]). In this example, the finish time of task 4 is 32.5 and the data transfer time from task 4 (on machine 0) to task 7 (on machine 2) is 8 (4 * 2 = 8) time units, hence the arrival time of the data from task 4 to task 7 is 40.5. The start time of task 7 is 45.5, therefore, the spare time between task 4 and task 7 is 5. This is the maximal value that the finish time of task 4



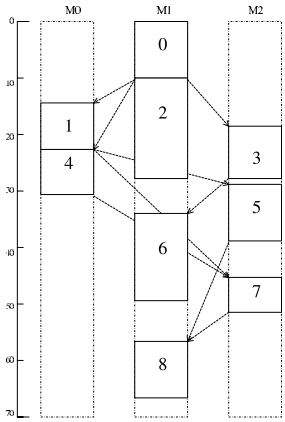
(a) an example graph

task	m0	m1	m2	task	m0	m1	m2
0	13	10	11	5	12	14	10
1	9	11	16	6	10	16	10
2	13	18	10	7	11	10	6
3	7	4	9	8	13	10	10
4	9	12	15				

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

machines	time for a data unit
m0 - m1	1.5
m1 - m2	1.0
m0 - m2	2.0

(c) communication cost between the machines



(d) the schedule derived by the HEFT algorithm

node	start time	nish time
0	0	10
1	14.5	23.5
2	10	28
3	19	28
4	23.5	32.5
5	29.5	39.5
6	34	50
7	45.5	51.5
8	57.5	67.5

(e) the start time and nish time of each node in (d)

Fig. 1. An example: the schedule is generated using the HEFT algorithm [19].

can be delayed at machine 0 without changing the start time of task 7.

In addition, for tasks i and j, which are adjacent in the execution order of a particular machine (and task i executes first), the spare time is defined as

$$\mathrm{Spare}_{\mathrm{SameMach}}(i,j) = ST(j) - FT(i),$$

where $FT(i)$ is the finish time of node i in the given schedule. In Fig. 1, for example, task 3 finishes at time 28, and task 5 starts at time 29.5; both on machine 2. The spare time between them is 1.5. In this case, if the execution time of task 3 delays for no more than 1.5 , the start time of task 5 will not be affected. However, one may notice that even a delay of less than 1.5 may cause some delay in the start time of task 6; to take this into account, we introduce one more parameter.

To represent the minimal spare time for each node, i.e., the maximal delay in the execution of the node that will not affect the start time of any of its dependent nodes (both on the DAG or on the machine), we introduce $MinSpare$, which is defined as

$$\mathrm{MinSpare}(i) = \min_{\forall j \in D_i} \mathrm{Spare}(i,j)$$

where D_i is the set of the tasks that includes the immediate successors of task i in the DAG and the next task in the execution order of the machine where task i is executed, and $Spare(i,j)$ is the minimum of $Spare_{DAG}(i,j)$ and $Spare_{SameMach}(i,j)$.

2.2. The slack of a node

In a similar way to the definition in [3], the slack of a node i is computed as the minimum spare time on any path from this node to the exit node. This is recursively computed, in an upwards fashion (i.e., starting from the exit node) as follows:

$$\mathrm{Slack}(i) = \min_{\forall j \in D_i} (\mathrm{Slack}(j) + \mathrm{Spare}(i,j)).$$

The slack for the exit node is set equal to

$$\mathrm{Slack}(i_{\mathrm{exit}}) = 0.$$

The slack of each task indicates the maximal value that can be added to the execution time of this task without affecting the overall makespan of the schedule. Considering again the example in Fig. 1, the slack of node 8 is 0; the slack of node 7 is also zero (computed as the slack of node 8 plus the spare time between 7 and 8, which is zero). Node 5 has a spare time of 6 with node 7 and a spare time of 9 with node 8 (its two immediate successors in the DAG and the machine where it is executing). Since the slack of both nodes 7 and 8 is 0, then the slack of node 5 is 6. Indeed, this is the maximal time that the finish time of node 5 can be delayed without affecting the schedule's makespan.

Clearly, if the execution of a task will start at a time which is greater than the statically estimated starting time plus the slack, the overall makespan (assuming the execution time of all other tasks that follow remains the same) will change. Our rescheduling policy is based on this observation and will selectively apply rescheduling based on the values of slack (or spare time). This is presented in the next section.

3. A selective rescheduling policy

The key idea of the selective rescheduling policy is to evaluate, at run-time, before each task starts execution, the starting time of each node against its estimated starting time in the static schedule and the slack (or the minimal spare time), in order to make a decision for rescheduling. The input of this rescheduler is a DAG, with its associated values, and a static schedule computed by *any* DAG scheduling algorithm. The objective of the policy is to optimize the makespan of the schedule while minimizing the frequency of rescheduling attempts.

As the tasks of the DAG are executed, the rescheduler maintains two schedules, S_1 and S_2. S_1 is based on the static construction of the schedule using estimated values; S_2 keeps track of what the schedule looked like for the tasks that have been executed (i.e., it contains information about only the tasks that have finished execution). Before each task (except the entry node) can start execution, its (real) start time can be considered as known. Comparing the start time that was statically estimated in the construction of S_1 and the slack (or the minimal spare time), a decision for rescheduling is taken. The algorithm will proceed to a rescheduling action *if* any delay between the real and the expected start time (in S_1) of the task is greater than the value of the *Slack* (or, in a variant of the policy, the *MinSpare*). This indicates that, in the first variant (*Slack*), the makespan is expected to be affected, whereas, in the second variant, the start time of the successors of the current task will be affected (but not necessarily the overall makespan). Once rescheduling is decided, the set of unexecuted tasks (and their associated information) and the already known information about the tasks whose execution has been completed (stored in S_2) are fed to the scheduling algorithm used to build a new schedule, which is stored in S_1. The values of *Slack* (or *MinSpare*), for each task, are subsequently recomputed from S_1. The policy is illustrated in Fig. 2.

4. Simulation results

4.1. The setting

To evaluate the performance of our rescheduling policy, we simulated both variants of our rescheduling policy (i.e., based on spare time and the slack) using four different DAG scheduling algorithms: Fastest Critical Path (FCP) [16], Dynamic Level Scheduling (DLS) [18], Hybrid Balanced Minimum Completion Time (HBMCT) [17], and Heterogeneous Earliest Finish Time (HEFT) [19]. Each algorithm generates the initial static schedule and is called again when the rescheduler decides to remap tasks.

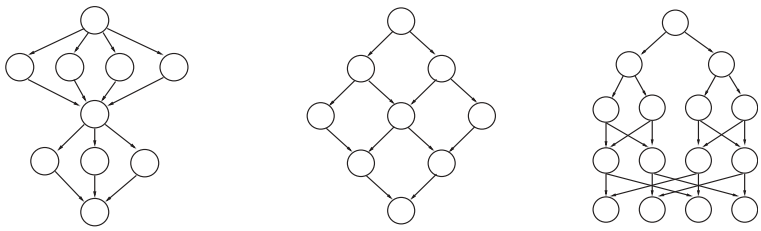
We have evaluated, separately, the behaviour of our rescheduling policy with each of the four different algorithms, both in terms of the performance of the final schedule and in terms of the running time. We used three different types of DAGs: FFT [12,19], Fork-Join Graphs [12], and Laplace [12]. Small-sized versions of each different type of DAG are shown in Fig. 3. Each of the resulting 12 experiments was carried out 100 times and average values were considered. In each case, we selected, randomly, the number of tasks in the DAG, and we generated a schedule using a number of machines randomly chosen between 3 to 8 (with equal probability). The static estimates for the execution of each task on each different machine are randomly generated from a uniform distribution in the interval [50,100], while the communication-to-computation ratio (CCR) is randomly chosen from the interval [0.1,1]. For the actual execution time of each task we adopt the approach in [6], and we use the notion of *Quality of Information* (QoI). This represents an upper bound

```
Input: an application graph G and a schedule S1 produced by an algorithm A
    (any algorithm for DAG scheduling onto heterogeneous systems may be used)

/* This variant makes use of the Slack value to decide whether to reschedule.
   Another variant could be based on MinSpare (in this case, all three occurrences
   of Slack below would be replaced by MinSpare). */

Selective rescheduling policy:
(1) Mark all tasks in S1 as unexecuted, Unexecuted[]
    S2 ← the real, post-execution schedule (initially empty)
(2) Compute for each task i from S1, Slack(i)
(3) While (Unexecuted[] is not empty)
      t ← first task in S1, which is in Unexecuted[] and whose input data are available
      m ← the allocated machine for t in schedule S1
      if (t is not the entry task in G)
         EST ← the expected start time of t in schedule S1
         RST ← the real start time of t on m in S2
         delay ← RST - EST
         if (delay > Slack(t))
            S1 ← A(Unexecuted[], S2)   /* reschedule remaining tasks */
            compute Slack for all tasks in S1, also in Unexecuted[]
            t ← first task in S1, which is in Unexecuted[]
            m ← the allocated machine for t in schedule S1
         endif
      endif
      execute task t on machine m
      S2 ← S2 ∪ {(t, m)}
      remove task t from the Unexecuted[] set
    endwhile
```

Fig. 2. The selective rescheduler.



(a) Fork-Join (b) Laplace equation solver (c) FFT

Fig. 3. Small-sized versions of 3 different types of DAGs.

on the percentage of error that the static estimate may have with respect to the actual execution time. So, for example, a percentage error of 10% would indicate that the (simulated) run-time execution time of a task will be within 10% (plus or minus) of the static estimate for the task. In our experiments we consider an error of up to 50%.

4.2. Scheduling performance

In order to evaluate the performance of our rescheduling policy, in terms of optimising the length of the schedule produced, we implemented both the spare time and the slack variants, and compared the schedule length they generate with three other approaches; these are denoted by *static*, *ideal*, and *always*. *Static* refers to the actual run-time performance of the original schedule (which was constructed using the static performance estimates); that is, no change in the original static schedule takes place at run-time. *Ideal* refers to a schedule, which is built *post mortem*; that is, the schedule is built *after* the run-time execution of each task is known. This serves as a reasonable lower bound

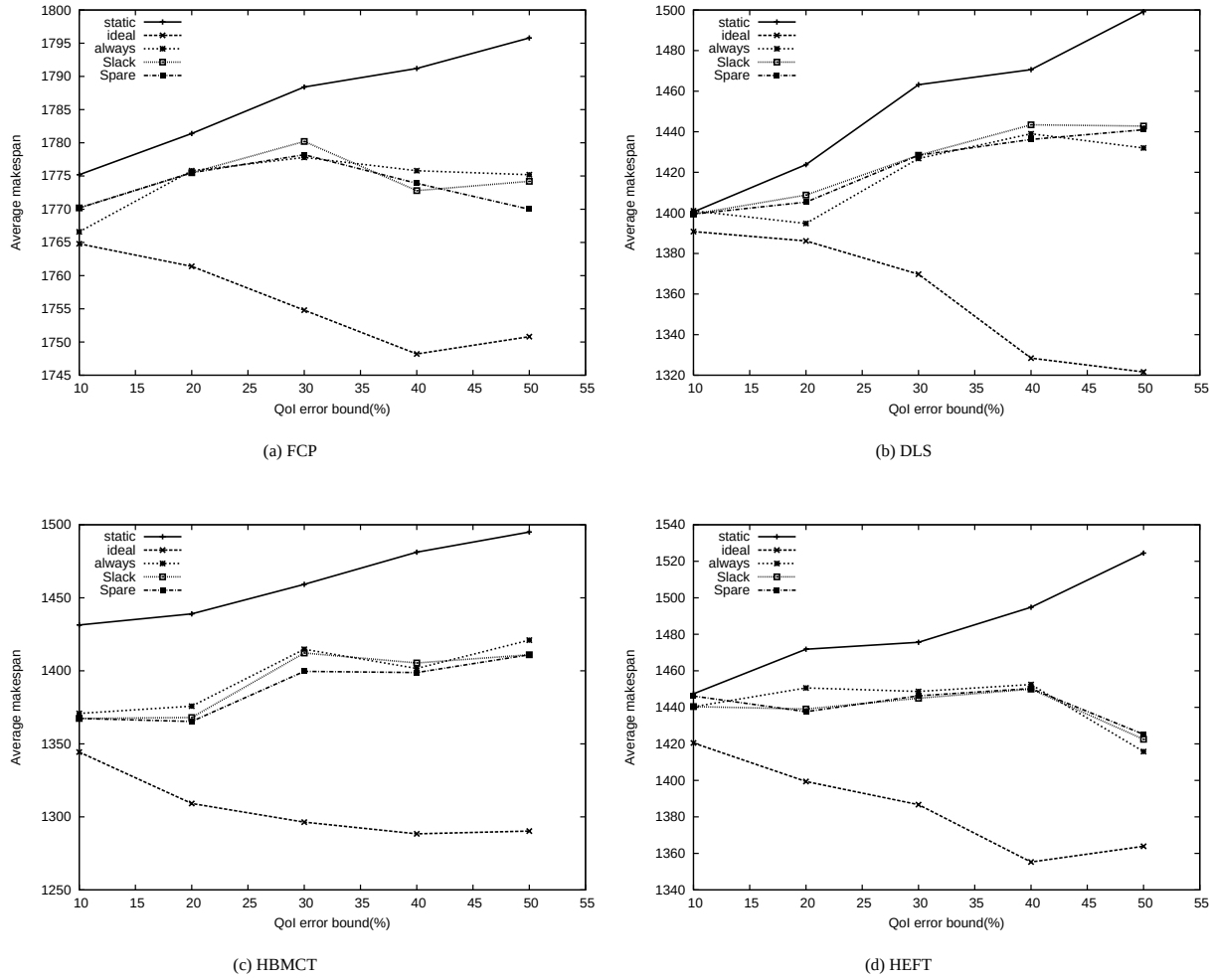


Fig. 4. Average makespan (over 100 runs on Laplace DAGs with 25–225 tasks and 3–8 machines) of four scheduling algorithms with dynamic rescheduling and our rescheduling policy.

to the makespan that rescheduling can achieve. Finally, *always* refers to a scheme that reschedules all remaining non-executed tasks each time a task is about to start execution.

The results, for each of the four different algorithms considered, and each different type of DAGs are shown in Figs 4–6. We considered a QoI percentage error between 10% and 50%. As expected, larger values of the QoI result in larger differences between the *static* and the *ideal*. The values of the three different rescheduling approaches (i.e., *always*, and the two variants of the rescheduling policy proposed in this paper, *slack*, *spare*) are roughly comparable. However, this is achieved at a significant benefit, since our policy attempts to reschedule only in a relatively small number of cases rather than always.

Another interesting remark from the figures is that rescheduling falls short of what can be considered to be the ideal time; this is in line with the results in [14]. The results also indicate that even for relatively high percentage errors, it is still the behaviour of the scheduling algorithm chosen that has the highest impact on the makespan. For instance, in all three types of DAGs, even the ideal makespan obtained with FCP is worse than the static makespan (i.e., no rescheduling), obtained with the other three scheduling heuristics.

4.3. Running time

Although the three rescheduling approaches that were compared in the previous section perform similarly, the approaches based on the policy proposed in this paper (i.e., *slack* and *spare*) achieve the same result

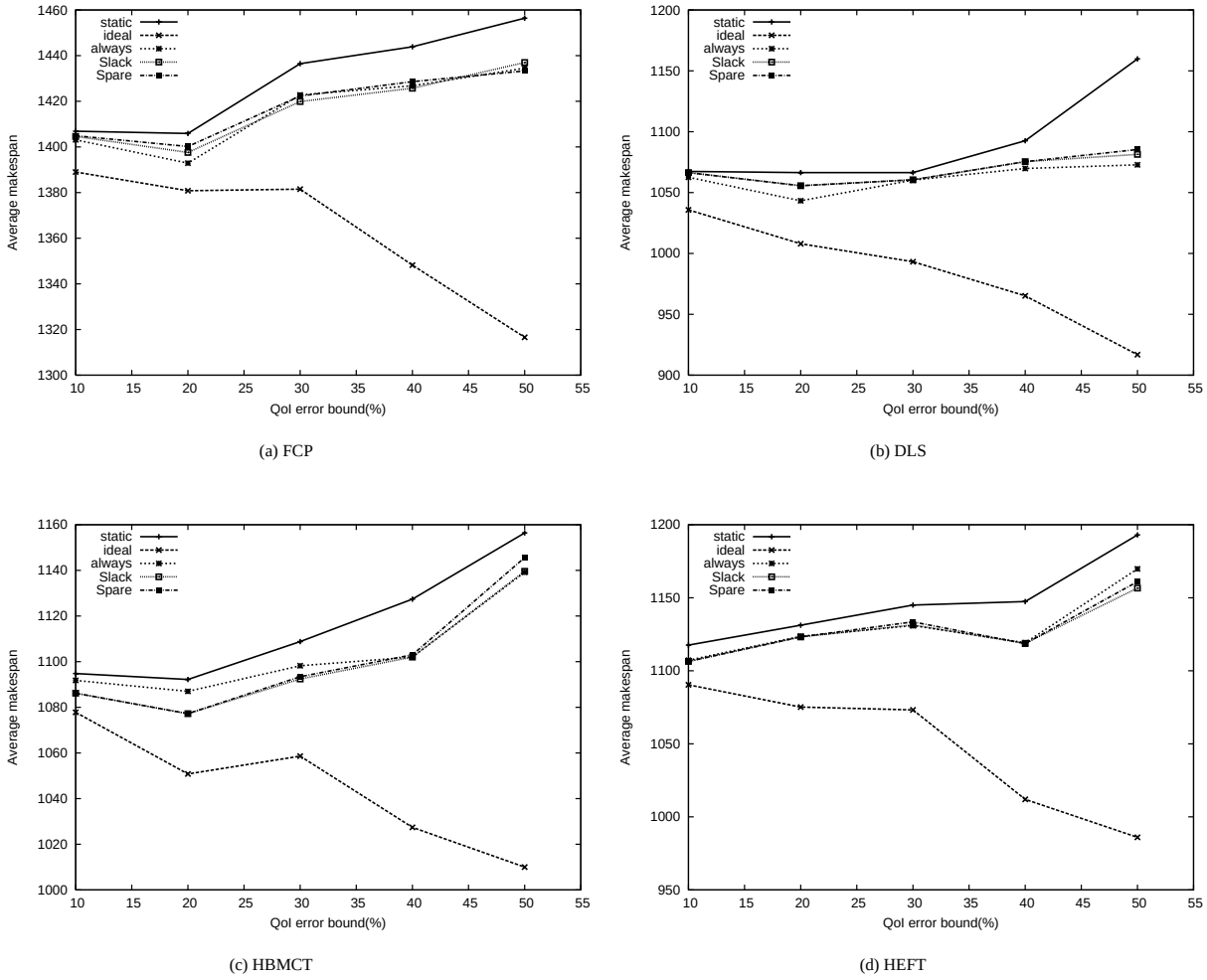
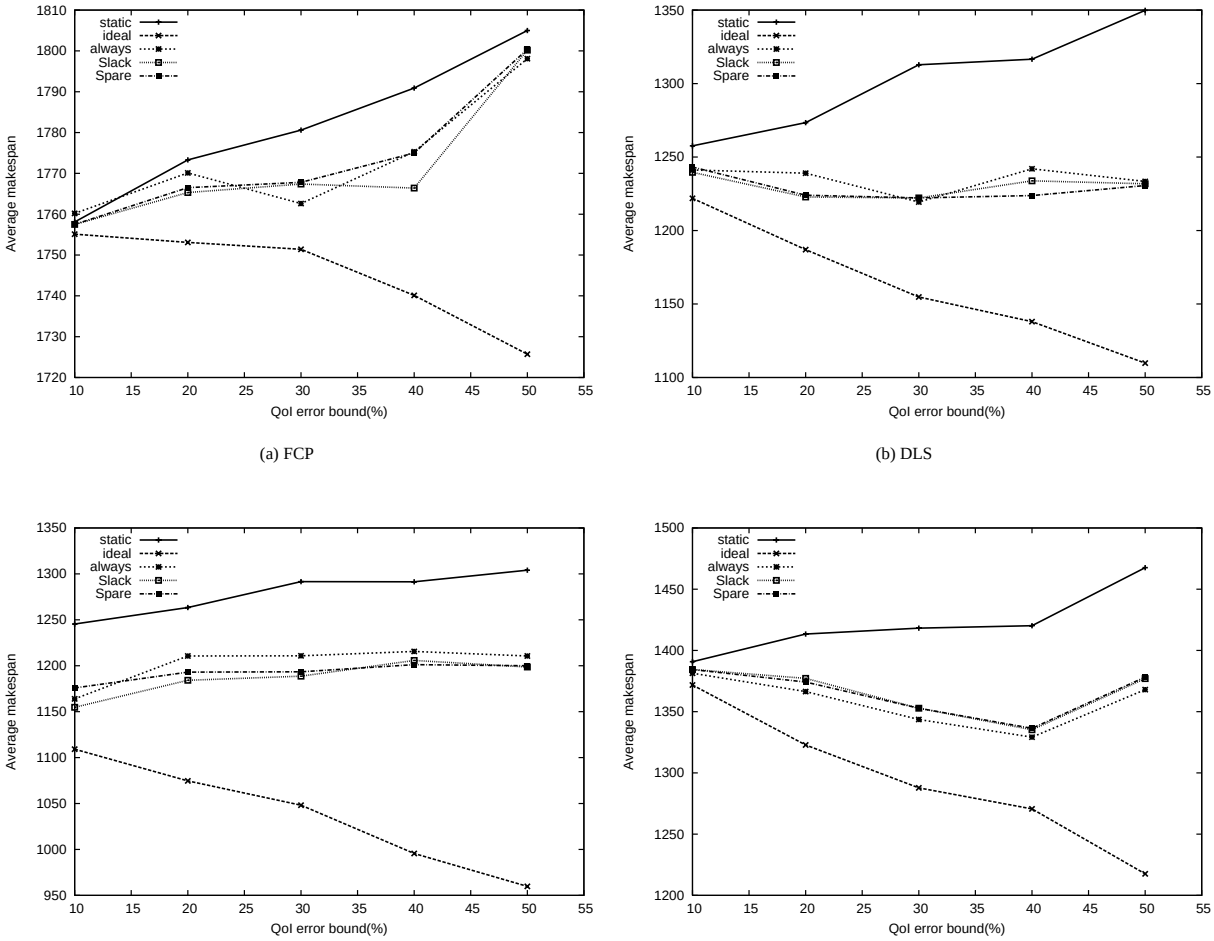


Fig. 5. Average makespan (over 100 runs on Fork-Join DAGs with 7–229 tasks and 3–8 machines) of four scheduling algorithms with dynamic rescheduling and our rescheduling policy.

(with *always*) at a significantly reduced cost. Table 1 shows the running time of each of the 3 approaches and for each different algorithm, averaged over 50 runs on all three types of DAGs with about 100 tasks each, using QoI 20%, and scheduling on 5 machines (column *R.T* in the table). It can be seen that the two variants of our policy run at no more than 43% of the time that is needed when rescheduling is performed after each task. Also, the two variants of our policy attempt to reschedule tasks at no more than 30% of the time (note that *always* would attempt to reschedule all the tasks except the entry node, hence the value of column *#R* in this case is equal to the number of tasks minus 1). Finally, it is interesting to notice that the number of tasks that are executed by a different machine than the one they were allocated to in the original static schedule appears to be dependent on the scheduling heuristic used and the type of DAGs considered (column *#C* in the table). In terms of algorithm performance, HEFT triggers rescheduling more times than the other three DAG scheduling algorithms. Furthermore, with either variant of our rescheduling policy, HBMCT appears to be resulting in fewer changes of the machine that would execute each task comparing to the static schedule (see column *#C*; especially visible in the case of Fork-Join DAGs). This is probably due to its good performance [17], an observation that would support an argument that those heuristics with good performance using statically estimated execution times appear to perform better also when there are run-time deviations from the static execution times.

Figure 7 shows how the running time varies if Fork-Join DAGs with up to 151 nodes are used. It can be seen that attempting to rescheduling always leads to



(a) FCP (b) DLS (c) HBMCT (d) HEFT

Fig. 6. Average makespan (over 100 runs on FFT DAGs with 15–223 tasks and 3–8 machines) of four scheduling algorithms with dynamic rescheduling and our rescheduling policy.

Table 1
Average values of running time (*R.T.*) in msec, number of times rescheduling is attempted (*#R*) and number of tasks that moved to another machine compared to the machine they were allocated to in the original static schedule (*#C*) for each of three rescheduling approaches using four algorithms. The average is calculated over 50 runs using 3 different types of DAGs each with around 100 tasks, QoI 20% and scheduling on 5 machines

		Always			Slack			Spare		
		R.T.	#R	#C	R.T.	#R	#C	R.T.	#R	#C
Laplace	HBMCT	3917.7	99	63.0	390.2	13.1	43.6	480.5	16.9	42.9
(100 tasks)	FCP	1862.9	99	38.8	256.0	10.7	39.3	333.0	13.8	45.5
	DLS	4971.4	99	72.1	393.7	12.1	40.3	568.0	17.4	51.1
	HEFT	1898.5	99	39.6	609.2	23.6	54.6	811.7	29.7	59.7
Fork-Join	HBMCT	4258.4	102	23.8	244.0	6.8	7.6	348.6	9.4	7.8
(103 tasks)	FCP	2075.5	102	39.7	411.5	9.8	41.3	483.9	11.6	49.6
	DLS	5684.8	102	32.4	366.4	8.3	12.3	446.9	10.5	13.4
	HEFT	2154.0	102	24.3	461.9	12.8	14.4	493.9	14.4	15.7
FFT	HBMCT	3546.0	94	39.1	361.7	16.0	24.6	392.9	17.5	23.5
(95 tasks)	FCP	1663.9	94	34.5	432.9	18.2	64.0	621.3	21.6	64.2
	DLS	4189.2	94	52.3	428.5	13.4	31.8	500.3	15.4	33.5
	HEFT	1706.6	94	36.3	543.8	22.0	59.0	563.1	23.1	61.3

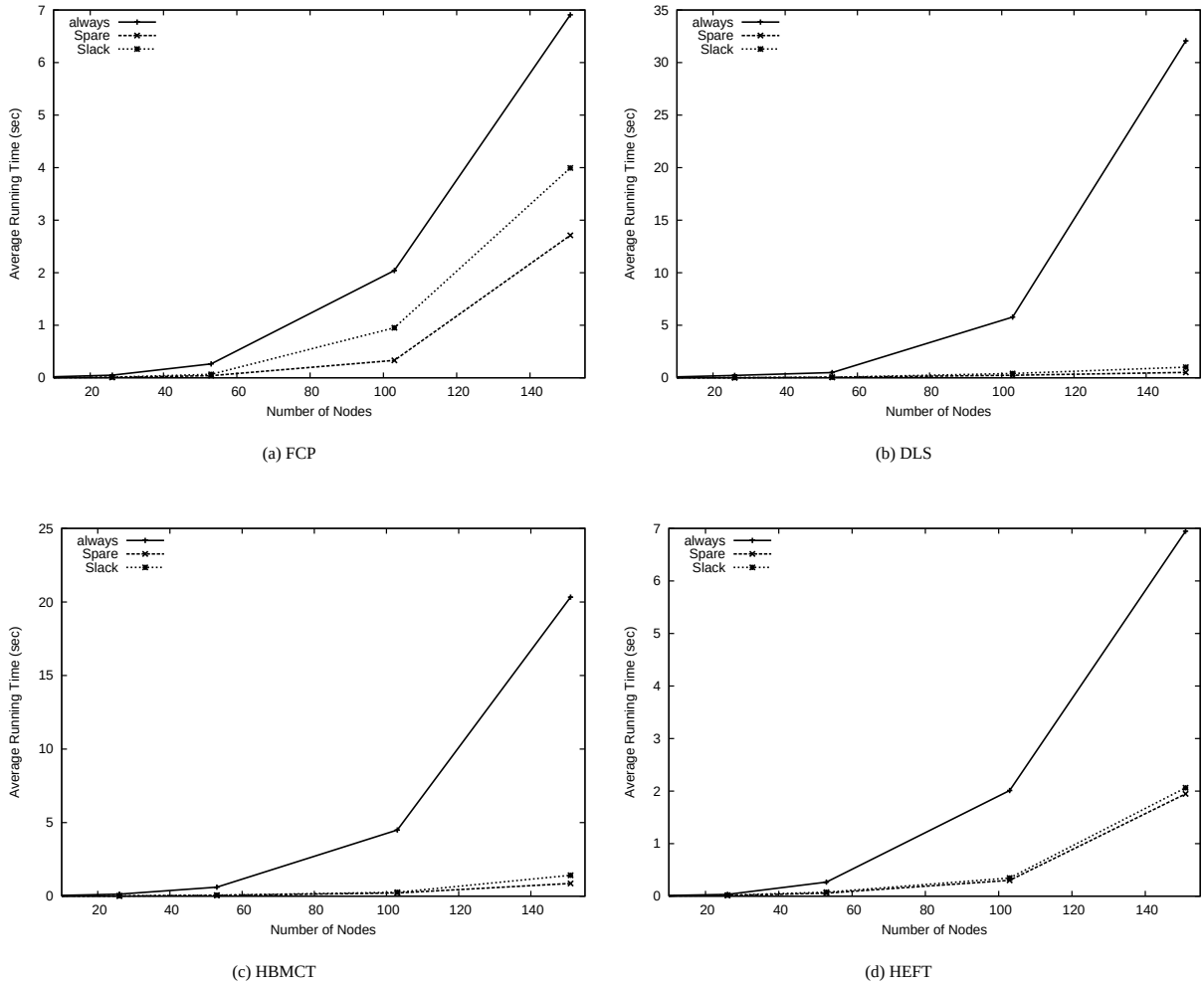


Fig. 7. Average running time (over 100 runs on Fork-Join DAGs with 7–151 tasks and 5 machines) of four scheduling algorithms with dynamic rescheduling and our rescheduling policy.

faster increases in the running time than our policy. It is worth noting that the slack variant is slightly faster than the spare variant; this is because the slack is cumulative and refers to the makespan of the schedule (as opposed to the spare time) and, as a result, it will lead to fewer rescheduling attempts (something that can also be observed from Table 1).

5. Conclusion

This paper presented a novel rescheduling policy for DAGs, which attempts to reschedule selectively (hence, without incurring a high overhead), yet achieving results comparable with those obtained when rescheduling is attempted for every task of the DAG. The approach is based on evaluating two metrics, the minimal spare time and the slack, and is generic, in that it can be applied to any scheduling algorithm.

Although there has been significant work in static scheduling heuristics, limited work exists in trying to understand how dynamic, run-time changes can affect a statically predetermined schedule. The emergence of workflows as important use cases in Grid computing as well as new ideas and approaches related to scheduling [13] are expected to motivate further and more elaborate research into different aspects related to the management of run-time information.

References

[1] S. Ali, A.A. Maciejewski, H.J. Siegel and J-K. Kim, *Definition of a Robustness Metric for Resource Allocation*, Proceedings of IPDPS 2003, 2003.

[2] F. Berman and R. Wolski, *The AppLeS project: a status report*, Proceedings of 8th NEC Research Symposium, Berlin, Germany, 1997.

[3] L. Boloni, and D.C. Marinescu, Robust scheduling of metaprograms, *Journal of Scheduling* **5** (2002), 395–412.

[4] R. Buyya, D. Abramson and J. Giddy, *Nimrod-G: an architecture for a resource management and scheduling system in a global Computational Grid*, in International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), Beijing, China.

[5] R. Buyya, J. Giddy and D. Abramson, *An evaluation of economy-based resource trading and scheduling on computational power Grids for parameter sweep applications*, Proceedings of the 2nd International Workshop on Active Middleware Service (AMS 2000), Kluwer Academic Press, 2000, 221–230.

[6] H. Casanova, A. Legrand, D. Zagorodnov and F. Berman, Heuristics for scheduling parameter sweep applications in Grid environments, Proceedings of the 9th Heterogeneous Computing Workshop (HCW'00), *IEEE Computer Society Press* (2000), 349–363.

[7] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi and M. Livny, Pegasus: Mapping Scientific Workflows onto the Grid, Proceedings of the 2nd AcrossGrids Conference, Cyprus, Springer-Verlag, *LNCS* **3165** (2004).

[8] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh and S. Koranda, Mapping Abstract Complex Workflows onto Grid Environments, *Journal of Grid Computing* **1** (2003), 25–39.

[9] J. Frey, T. Tannenbaum, I. Foster, M. Livny and S. Tuecke, Condor-G: a computation management agent for multi-institutional Grids, *Journal of Cluster Computing* **5** (2002), 237–246.

[10] A. Hoheisel and U. Der, An XML-Based Framework for Loosely Coupled Applications on Grid Environments, Proceedings of ICCS 2003, Springer-Verlag, *LNCS* **2657** (2003), 245–254.

[11] H. Hoschek, J.J. Martinez, A. Samar, H. Stockinger and K. Stockinger, *Data management in an international Data Grid project*, Proceedings of the First IEEE/ACM International Workshop on Grid Computing, 2000, pp. 77–90.

[12] Y.K. Kwok and I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, *Journal of Parallel and Distributed Computing* **59** (1999), 381–422.

[13] J. MacLaren, R. Sakellariou, K.T. Krishnakumar, J. Garibaldi and D. Ouelhadj, *Towards Service Level Agreement Based Scheduling on the Grid*, Proceedings of the Workshop on Planning and Scheduling for Web and Grid Services, 2004, pp. 100–102.

[14] M. Maheswaran and H.J. Siegel, *A dynamic matching and scheduling algorithm for heterogeneous computing systems*, Proceedings of the 7th Heterogeneous Computing Workshop (HCW'98), 1998, pp. 57–69.

[15] 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) (June 2002), 648–658.

[16] A. Radulescu and A.J.C. van Gemund, *On the complexity of list scheduling algorithms for distributed memory systems*, Proceedings of the 13th ACM International Conference on Supercomputing, ACM Press, 1999, 68–75.

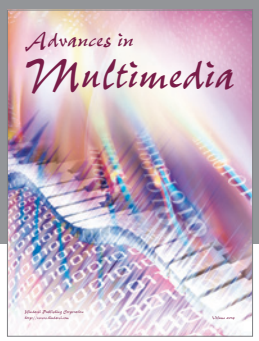
[17] R. Sakellariou and H. Zhao, *A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems*, Proceedings of the 13th Heterogeneous Computing Workshop (HCW'04), 2004.

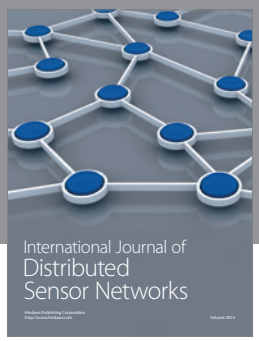
[18] G.C. Sih and E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architecture, *IEEE Transactions on Parallel and Distributed Systems* **4**(2) (February 1993), 175–187.

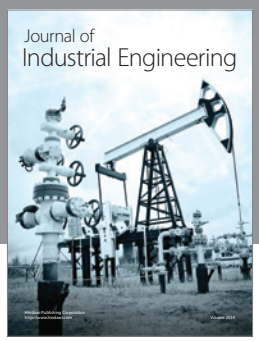
[19] H. Topcuoglu, S. Hariri and M. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, *IEEE Transactions on Parallel and Distributed Systems* **13**(3) (March 2002), 260–274.

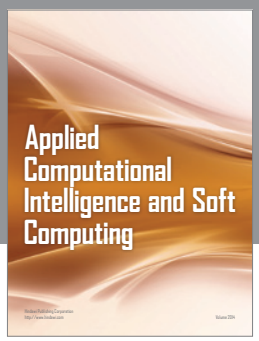
[20] H. Zhao and R. Sakellariou, An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm, Proceedings of Euro-Par 2003, Springer-Verlag, *LNCS* **2790** (2003), 189–194.

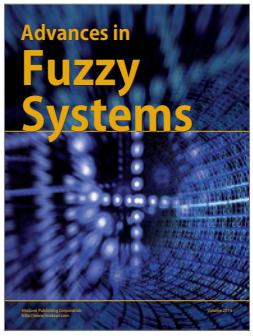
[21] H. Zhao and R. Sakellariou, A Low-Cost Rescheduling Policy for Dependent Tasks on Grid Computing Systems, Proceedings of the 2nd Across Grids Conference, Springer-Verlag, *LNCS* **3165** (2004).

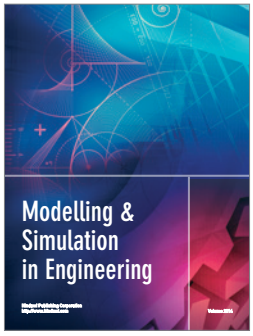


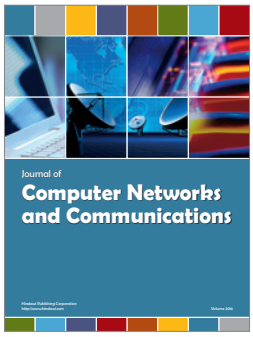












Hindawi

Submit your manuscripts at
http://www.hindawi.com