A Low-Cost Rescheduling Policy for Dependent Tasks on Grid Computing Systems

Henan Zhao and Rizos Sakellariou

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

Abstract. 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. To deal with this, 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 in the execution. Yet, 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]. This model provides an easy way of addressing the mapping problem; a schedule is built by assigning the nodes (the term task is used interchangeably with the term node throughout the 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 [13, 16] 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, real-time variations may affect the schedule's performance significantly.

An obvious response to changes that may occur at run-time is to reschedule, or readjust the schedule dynamically, using additional information that becomes

available at run-time. 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 [7], Data Grid [9] 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 [12]. 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, 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 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 preliminary simulation experiments, this policy gives equally good performance with policies that consider for rescheduling every task of the DAG, at a much lower cost; in our experiments, SR considers less than 20% of the tasks of the DAG for rescheduling.

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 and, 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 is minimized. For an example, see Figure 1, and parts (a), (b), and (c).

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 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

$$Spare_{DAG}(i,j) = ST(j) - DAT(i,j),$$

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 Figure 1 and the schedule in Figure 1(d) (derived using the HEFT heuristic [16]). 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 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

$$Spare_{SameMach}(i,j) = ST(j) - FT(i),$$

where $FT(i)$ is the finish time of node i in the given schedule. In Figure 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

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

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:

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

The slack for the exit node is set equal to

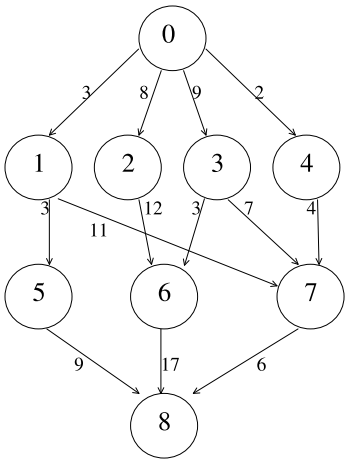
$$Slack(i_{exit}) = 0.$$

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 Figure 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 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 (and the 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



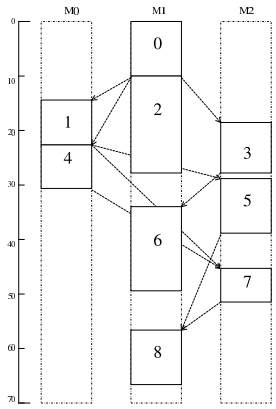
(a) an example graph

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	finish 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 finish time of each node in (d)

Fig. 1. The schedule generated by the HEFT algorithm

estimated starting time in the static schedule and the slack (or the minimal spare time) 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* 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 ex-

```
Input: an application graph G and a schedule S1 produced by an algorithm A

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) // (or MinSpare(i))
(3) While (Unexecuted[] is not empty)
      t ← first task in S1, which is in Unexecuted[]
      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)) // or (delay > MinSpare(t))
            S1 ← A(Unexecuted[], S2) // reschedule
            compute MinSpare for all tasks in S1, also in Unexecuted[] // or Slack
            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)}
      Unexecuted[] ← Unexecuted[] \ t
    endwhile
```

Fig. 2. The Selective Rescheduler.

ecution, 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 a 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*) are subsequently recomputed from S_1.

The policy is illustrated in Figure 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) [14], Dynamic Level Scheduling (DLS) [15], Heterogeneous Earliest Finish Time (HEFT) [16] and Levelized-Min Time (LMT) [10]. Each algorithm provides 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 randomly generated DAGs, each consisting of 50 to 100 tasks, following the approach described in [17], and we tried to schedule them on 3 to 8 machines (randomly chosen with equal probability for each machine). The estimated execution of each task on each different machine is 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 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 to the performance that rescheduling can achieve. Finally, *always* refers to a scheme that re-schedules all remaining non-executed tasks each time a task is about to start execution.

The results, for each of the four different algorithms considered, are shown in Figure 3. We considered a QoI error percentage from 10% to 50 %. As expected, larger values of the QoI error 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 policy proposed in this paper, *slack*, *spare*) are roughly comparable. However, this is achieved at a significant benefit, since our policy

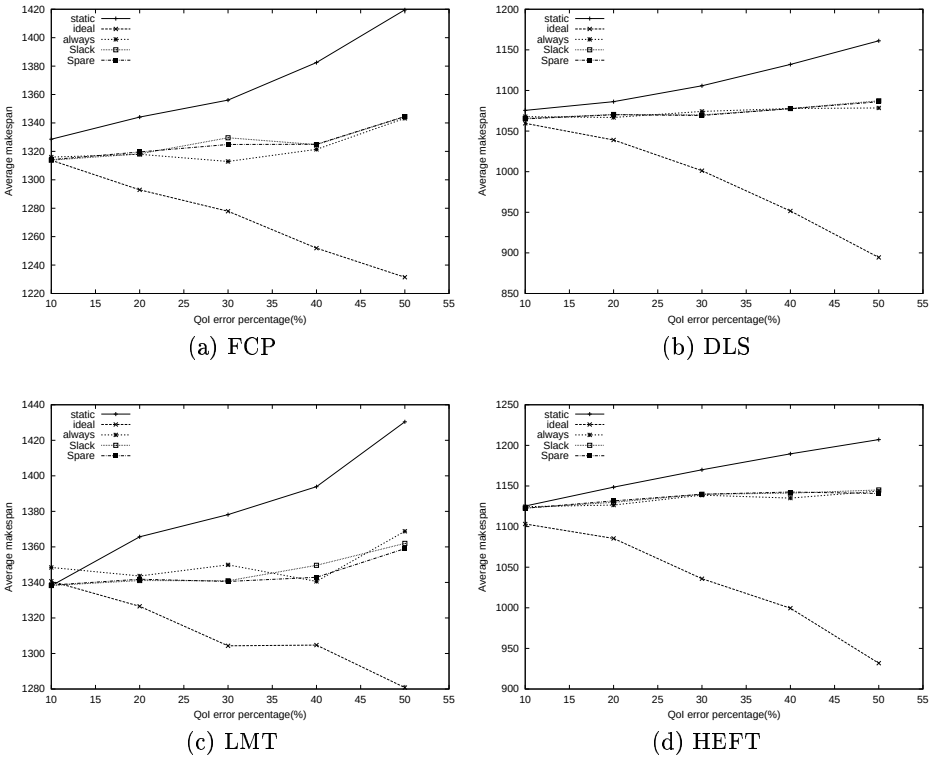


Fig. 3. Average makespan (over 100 runs on randomly generated DAGs) for various levels of QoI with four scheduling algorithms.

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 assumed to be the ideal time; this is in line with the results in [12]. 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.

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 (with *always*) at a significantly reduced cost. Table 1 shows the running time of each of the 3 approaches averaged over 50 runs on DAGs of 50 tasks each, using QoI 20%, and scheduling on 5 machines. It can be seen also that the two variants of our policy run at no more than 25% of the time that is needed and attempt to reschedule tasks at no

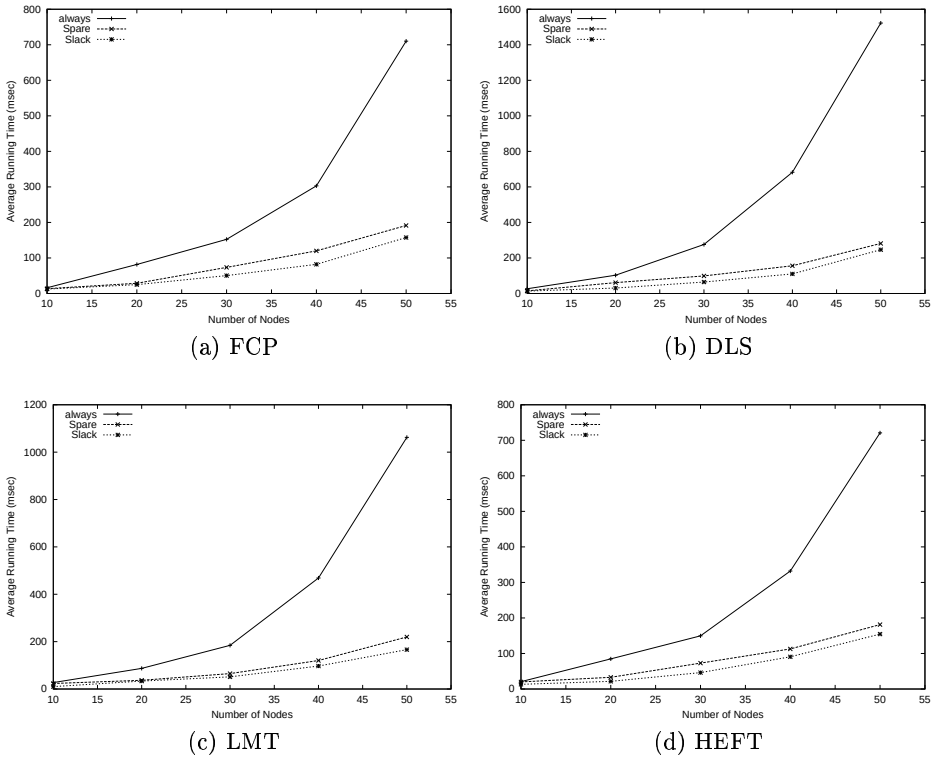


Fig. 4. Average running time (over 100 runs on randomly generated DAGs with fixed 5 machines) of four scheduling algorithms with dynamic scheduling and our rescheduling policy.

more than 20% of the total number of tasks (note that *always* would attempt to reschedule all the tasks except the entry node, hence the value 49). Figure 4 shows how the running time varies if DAGs having 10 to 50 nodes are used. It can be seen that attempting to rescheduling always leads to 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.

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

	Always		Slack		Spare	
	R.T.	*#R*	*R.T.*	*#R*	*R.T.*	*#R*
FCP	345.77	49	66.87	6.82	74.28	7.83
DLS	699.33	49	122.18	7.35	126.23	7.95
LMT	528.23	49	77.93	6.51	97.15	8.41
HEFT	357.40	49	73.01	7.51	86.20	8.86

Table 1. Average running time and number of times rescheduling is attempted for each of three rescheduling approaches using four algorithms. The average is over 50 runs using randomly generated DAGs each with 50 tasks, QoI 20% and scheduling on 5 machines.

minimal spare time and the slack, and is general, 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 important use cases in Grid computing, such as workflows, as well as new ideas and approaches related to scheduling [11] 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. In *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. In *Journal of Scheduling*, 5:395-412, 2002.
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. In *2nd International Workshop on Active Middleware Service (AMS 2000)*, USA, 2000.
6. 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.
7. 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*, 5:237-246, 2002.
8. A. Hoheisel and U. Der. An XML-Based Framework for Loosely Coupled Applications on Grid Environments. Proceedings of *ICCS*, 2003 (to appear).
9. 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*, India, 2000.

10. M. Iverson, F. Ozguner, and G. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *Heterogeneous Computing Workshop*, pp. 93-100, 1995.
11. J. MacLaren, R. Sakellariou, J. Garibaldi and D. Ouelhadj. Towards Service Level Agreement Based Scheduling on the Grid. Proceedings of the *2nd Across Grids Conference*, Cyprus, 2004.
12. M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *7th Heterogeneous Computing Workshop(HCW'98)*, March 1998.
13. 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.
14. A. Radulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed memory systems. In *ACM International Conference on Supercomputing*, 1999.
15. 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.
16. 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.
17. H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par 2003*. Springer-Verlag, LNCS 2790, 2003.