RE-SCHEDULING AND ERROR RECOVERING ALGORITHM FOR DISTRIBUTED ENVIRONMENTS

Alexandra OLTEANU[1], Florin POP[2], Ciprian DOBRE[3], Valentin CRISTEA[4]

Planificarea eficientă este cheia performanței aplicațiilor Grid, aspect ce solicită asigurarea toleranței la defecte şi a recuperării din erori. Soluțiile pentru toleranța la defecte în planificarea în sistemele Grid, considerate în această lucrare, se bazează pe recuperarea erorilor şi re-planificări controlate. Acest articol propune un nou algoritm pentru re-planificare generică şi recuperare din eroare, conceput pe baza metodelor euristice de planificare existente, care sunt alese în prealabil în funcție de structura sistemului distribuit în care se face planificarea. Componenta de re-planificare este simulată. În acest mediu simulat este folosit un monitor care interoghează periodic sistemul de resurse pentru a determina starea acestuia. Rezultatele experimentale arată că strategia propusă asigură toleranță la defecte pentru procesul de planificare în medii distribuite.

Scheduling is the key to Grid applications performance, but for everything to work fine we need to ensure the fault tolerance. The fault tolerant solutions for Grid scheduling, considered in this paper, are based on error recovery and rescheduling. This paper proposes a novel generic rescheduling concept, which allows the use of the designed rescheduling algorithm with a wide variety of scheduling heuristics which are chosen in advance depending on the system structure. The rescheduling component is called by a monitor who interrogates the system periodically. Furthermore, the experimental results show that the proposed strategy ensure fault tolerance.

Keywords: Grid Scheduling, Fault Tolerance, Error Recovering, Simulation

1. Introduction

Due to the NP-complete characteristic of scheduling problem in its general form for distributed environment, a number of heuristics have been proposed. Scheduling is the process of allocating a set of resources to tasks or jobs to achieve certain performance objectives satisfying certain constraints. In addition,

[1] MCS Student, Computer Science Department, University POLITEHNICA of Bucharest, Romania, e-mail: alexandra.olteanu@cti.pub.ro
[2] Lecturer, Computer Science Department, University POLITEHNICA of Bucharest, Romania, e-mail: florin.pop@cs.pub.ro
[3] Lecturer, Computer Science Department, University POLITEHNICA of Bucharest, Romania, e-mail: ciprian.dobre@cs.pub.ro
[4] Professor, Computer Science Department, University POLITEHNICA of Bucharest, Romania, e-mail: valentin.cristea@cs.pub.ro

for providing fault tolerance to the system, they monitor the progress of application execution, and perform rescheduling or another fault tolerance mechanism if any abnormal behavior of the resources in the schedule is detected.

Fault tolerance is an important propriety for distributed computing in Grid systems where the resources availability cannot be guaranteed. The most common and easiest to detect errors are those related to timing, omissions and interactions.

The main objective of the present work is to develop an algorithm for rescheduling and error recovery that will be able to ensure transparency in large distributed computing systems, where are data dependencies among tasks that must be scheduled and executed. To test the developed rescheduling and error recovering algorithm, it was integrated into the simulator MONARC, considering the details of its architecture. The validation of concepts introduced by the algorithm was done by simulation. For this to be possible, MONARC simulator was extended with a set of components for the simulation of errors occurrence and monitoring.

The paper presents in Section 2 the related work regarding rescheduling in distributed environment. Section 3 presents a brief overview of fault tolerance in grid environments, then, in Section 4 the re-scheduling and error recovering algorithm is described. Section 5 highlights the MONARC extension and testing scenarios. In Section 6 the experimental results are presented and analyzed. The conclusions and future work are presented in Section 7.

2. Related work

Rescheduling is associated in the literature with two approaches: for optimizing the scheduling performance, to obtain a better finalization time, and for providing fault tolerance. A rescheduling policy, proposed in [2], considers rescheduling at some carefully selected points along execution. After the initial schedule is made, it selects some jobs for rescheduling, if the run time performance variation exceeds a predefined threshold. Other papers present rescheduling strategies based on a single scheduling heuristic by which the workflow planner can adapt to the grid dynamic changes. Most evaluations and analysis studies of various heuristics, surprisingly, showed that similar values are obtained for results quality, which identifies the same strengths and weaknesses, the differences being given only by few percent [2]. For these reasons, in this work, was chosen to implement a generic scheduling algorithm, which can be associated with almost any scheduling heuristic.

Scheduling is the decision process that assigns tasks of the bag-of-tasks applications to available resources in order to optimize various performance metrics. This presentation is mainly based on the scheduling workflow, in which the main job is divided into several tasks connected by data-dependences. The

main job is represented by a DAG (directed acyclic graph), in which graph nodes represent tasks and graph edges represent data transfers. A node in graph must be assigned to a resource in Grid, and communication that is represented by graph edge must appear in the network between resources to which the nodes are assigned. Another important consideration is related to distinguish two categories in which the scheduling problem can be divided [3]: scheduling independent task, which are schedule just to increase the total system performance, and scheduling tasks with dependencies that implies the assignment of tasks without violating the precedence constrains in order to minimize the deadline on a distributed system.

MCP (Modified Critical Path) - algorithm based on lists with two phases: the prioritization and selection of resources. Parameter used to prioritize nodes is ALAP (As Late As Possible).

CCF (Cluster ready Children First) – dynamic scheduling algorithm based on lists. In this algorithm the graph is visited in topological order, and tasks are submitted as soon as scheduling decisions are taken. The algorithm considers that when a task is submitted for execution it is inserted into the RUNNING-QUEUE. If a task is extracted from the RUNNING-QUEUE, all its successors are inserted into the CHILDREN-QUEUE. The running ends when the two queues are empty.

ETF (Earliest Time First) - algorithm based on keeping the processors as busy as possible. It computes, at each step, the earliest start times of all ready nodes and selects the one having the smallest start time.

HLFET (Highest Level First with Estimated Times) - use a hybrid of the list-based and level-based strategy. The algorithm schedules a task to a processor that allows the earliest start time.

Hybrid Remapper PS (Hybrid Remapper Minimum Partial Completion Time Static Priority) – is a dynamic list scheduling algorithm specifically designed for heterogeneous environments. The set of tasks is partitioned into blocks so that the tasks in a block do not have any data dependencies among them. Then the blocks are executed one by one.

3. Fault tolerance in Grid Environments

Important components of a fault-tolerant system are the identification and classification of the types of errors that can occur in order to find clues about how to implement or improve error detection and recovering. Grid computing environments are particularly prone to defects because [1]: these systems are composed of a wide variety of services, software and hardware components that need to interact with each other. System failures can result not only because of an error on a single component, but also from interaction between components; these systems are highly dynamic, with components that enter and exit the system all

the time; the probability of errors occurrence is amplified by the fact that many Grid applications composed of tasks that last a long time.

When determining the types of errors that we could consider the following class [4][5][6]:

network errors	Environmental errors caused by communication channel and basically refer to package losses on the transmission path or corrupted incoming packages on the receiving path
timing errors	Formed from two types of errors depending on the time of their appearance: at the beginning of the connection or later. The first category is due to the inability to establish a connection, and the second, which is considered a performance error, occurs when the response time exceeds the time in which the receiver expects to receive a reply
response errors	Caused by a service which returns values outside of the expected boundaries by the caller or it refers to errors that appear at the transition between system states.
omission errors	messages are delayed or lost
physical errors	this includes CPU errors, memory errors, storage errors, etc
life cycle errors	particular to components which expose services which can expire at a certain moment
interaction errors	caused by incompatibilities at the communication protocol stack level, security, workflows or timing
byzantine errors	arbitrary errors that could appear during the execution of an application

Many works in the field led to the description of fault tolerance mechanisms for applications, for preserving application execution despite the presence of a processor fails. These mechanisms are classified into two major categories according the level at which errors are treated. The first category is at the task level, in which information about the task is sufficient to redefine the status of a failed task. The second category is at the application level, where much more information is necessary to redefine the entire state of application. According to [7], in the category of task level mechanism we distinguish the strategies: retry, alternate resource, and checkpoint and task duplication. After detecting the failure the retry approach simply considers a number of tries to execute the same task on the same resource. The checkpoint/restart approach saves the computation state periodically, such that it migrate the saved work of failed tasks to other processors. The alternate resource strategy chooses another resource for executing the tasks on the case of task failure. The task duplication mechanism selects tasks for duplication, hoping that at least of the replicated tasks will finish successfully.

In the category of application level mechanisms we have the strategies: rescue file, redundancy, user-defined exception handling and rewinding. The rescue file mechanism consists of the resubmission of uncompleted portions of a DAG when one or more tasks resulted in failure. The user-defined exception handling allows users to give a special treatment to a specific failure of a particular task. The rewinding mechanism seeks to preserve the execution.

4. Re-scheduling and Error Recovering Algorithm

When an error occurs at one of the scheduled tasks, it's coming out the necessity of rescheduling it. Considering the task belonging to a directed acyclic graph (DAG) we must analyze if the node where the error occurred can be reschedule alone or we should reschedule it with all the others nodes that forms together a dependency sub-graph having root the current node.

Based on this assumptions there are two types of input data in terms of the set of tasks: one task or tasks sub-graph. Other input data that should be provided to the algorithm are: scheduling heuristic that should be used and the system set of the available resources. After rescheduling, the tasks execution order is reordered and new associations (task, CPU) are built and sent to execution. This represents output data that the rescheduling algorithm returns.

To easily insert the rescheduling fault tolerance mechanism in any type of systems, we have designed a rescheduling algorithm that can be used in combination with a wide variety of scheduling algorithm which are chosen in advance depending on the system structure (here we may consider factors like the number of existing processors, the structure of graph task that we want to schedule) to achieve optimal results. The proposed generic rescheduling algorithm is described below:

```
H – heuristic used for rescheduling
P – scheduling
R – available resources
Initialize schedule P_current= initial schedule of DAG
While (DAG unfinished)
     If( error detected)
          Update R
          P= schedule(P_current,H)
          If (P_0.task_asoc_to_res!=P_1.task_asoc_to_res)
               P_current =P
               Execute P_current
```

The schedule procedure call a scheduling algorithm (HLFET, CCF, MCP, ETF, HybridRmapper) to schedule the graph section remained unfinished, pointed by the heuristic H.

5. The use of MONARC Simulator

The MONARC 2 is a simulator for large scale distributed systems, having as a purpose the modeling and simulation of distributed systems, with the goal of predicting general performances of the applications running on these systems. MONARC is built based on a process oriented approach for discrete event simulation, which is well suited to describe concurrent running programs, network traffic [6]. For modeling the rescheduling mechanism for a DAG in case of errors occurrence, the MONARC simulator had been extended with simulation components for error occurrence, catalog of states, monitoring and rescheduling, which are highlighted in Fig. 1.

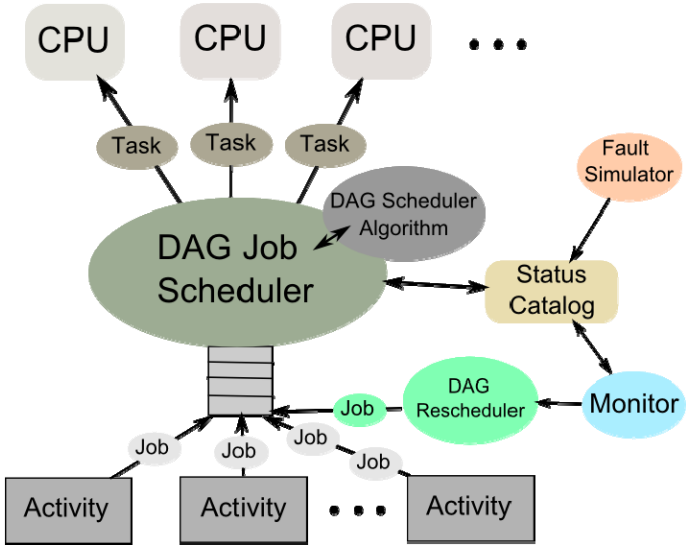

Fig.1. The way in which the new simulator components interacts with old ones

First, the default behavior of the simulator does not take into account the situation when the DAG task fails, and because of this assumption do not have implemented an error simulation mechanism. Therefore, we extended the default behavior of the simulator that can simulate the appearance of errors by implementing a catalog of states of tasks and a method for error simulation by setting a task to the error status. Further, to keep fault tolerance for the simulator, we added a monitoring component that periodically analyzes the states catalog and if it finds a node in the error state it calls another component, which

represents the main objective of this paper, the rescheduling components. The rescheduling component decide to act based on the information provided by the node that is found in error state, analyses and then decide the rescheduling strategy to be used for each case. After determining the rescheduling strategy, we can have two situations: reschedule a single node or building a sub graph which is sent to the scheduler as a DAGJob for rescheduling. Rescheduling and monitoring are triggered when the job (DAGJob) is submitted for the first time in the system.

For a closer simulation of real environmental behavior, the simulator should have also included an error simulation component. To implement this, for sets of tasks mapped on a DAG, was built a catalog that contains the running tasks status: created, submitted, running, finished and error. This offers information about the graph progress status, i.e. information about the successfully finalized tasks, about tasks where error occurs and about those tasks found in execution. Based on the information provided by this catalog at any given time will decide how to make the rescheduling if exist tasks in error state. How one can move from one state to another is shown in Fig. 6.1.

For error simulation are implemented two methods that determine the rate, i.e. the minimum number of errors that may exist in the system at a given time. In this way it can decide if the simulated system is fault-prone or not, and to what extent. Also using the catalog one can check, close or open, the monitor, which in case of error occurrence call the rescheduling procedure. When a node that is in error is required for rescheduling, the graph is traversed in topological order and the first node found is returned.

Rescheduling component implemented for the MONARC simulator is called by the monitor in case of error detection. It receives the node where the error occurred and, after deciding if the node is an inner node or an edge node, calls the rescheduling strategies which are predefined for each case. Choosing an appropriate strategy is very important because it can influence the rescheduling performance.

The rescheduling algorithm, previously presented, was implemented for the MONARC simulator:

- monitor implements the iteration and check if nodes are in the error state
- “scheduling” procedure chooses the rescheduling strategy, considering the node type, and sends for rescheduling the node or the sub graph.

6. Tests scenarios and experimental results

The purpose of simulation is to check the quality of scheduling algorithms. Algorithms evaluation is made using a set of tests. For testing we used two different sets of configurations, described by two configuration files:

- Set 1: 3 processors and a set of 14 tasks
- Set 2: 50 processors and sets of 25/50/100/500 tasks

On the first configurations set would be made comparisons between the algorithms performances considering several metrics. To make a pertinent comparison of how each algorithm behaves in case of rescheduling, for these tests we have forced crashes on the same graph node.

The second set of tests is used to test the performance of rescheduling algorithm in case of larger graphs: 25, 50, 100 and 500 nodes. For this, error simulation was made limiting the number of errors at some time to 5% of the number of nodes. The simulation of error occurrence is made and several steps through which the graph passes from the moment when it is sent to execution in a virtual distributed environment, provided by the MONARC simulator, are highlighted in the Figs. 2 and 3.

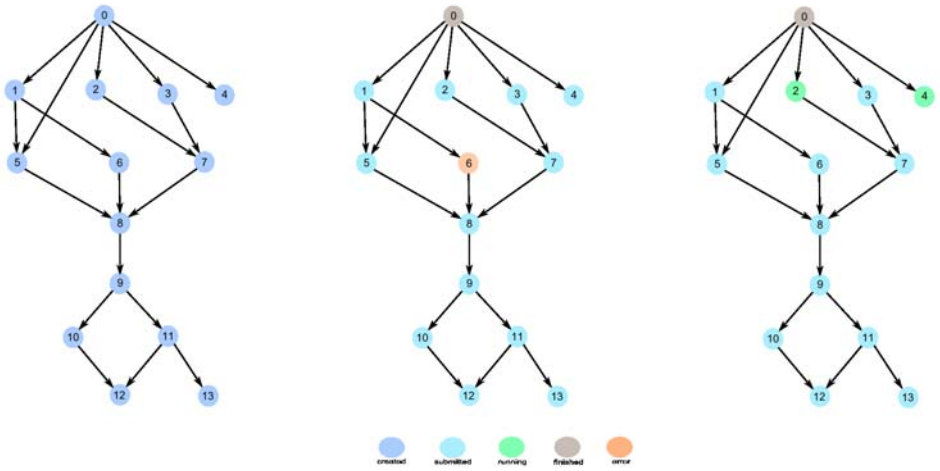

Fig. 2. A simple DAG example and a few steps in the rescheduling algorithm execution, using MCP heuristic

The states in which the nodes can be are: created, submitted, running, finished, error. In the first graph in Fig. 2 all nodes are created and start their execution. In the second graph we can observe that node 0 is finalized and at node 6 an error had occur, the rest of the graph being only submitted. So the sub graph consisting of node 6 and all nodes that depend on its execution are rescheduled

and switched to “submitted” state. In the last graph in Fig. 7.1 the reschedule was made and nodes 2 and 4 are running.

In the first graph in Fig. 3 is observed that nodes 1, 2, 3, 4 and 6 are completed, node 7 is running, node 13 is in “error” state and the remaining nodes are in “submitted” state. Node 13 is rescheduled alone because is an edge node. In the next graph nodes from 0 to 8 are completed, node 9 is running, the nodes 11, 12 and 13 are in “submitted” state and node 10 is in “error” state. Node 10 with node 12, which depends on node 10 execution, are rescheduled. In the last graph all nodes are completed.

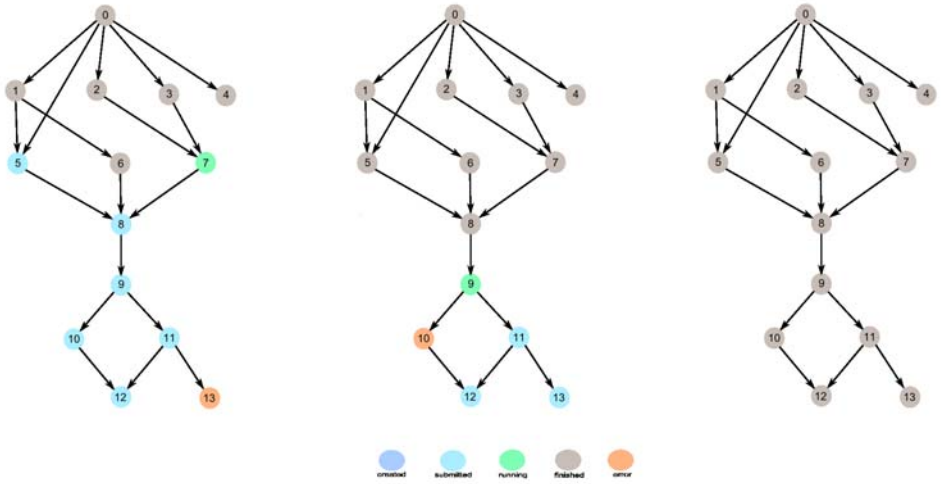

Fig. 3. A simple DAG example and a few steps in the rescheduling algorithm execution, using MCP heuristic

On the example graph that had been illustrated will be presented the simulation results for the scheduling algorithms using a set of metrics (execution time, number of running tasks per unit time, number of completed tasks per unit time), for both test scenarios, with and without errors occurrence.

Generating test data was made both by designing a small size graph, presented in the previous chapter and used to compare the performance between different scheduling algorithms using a graph generator to generate graphs with 20, 50, 100 and 500 nodes.

We analyze the execution time obtained for each experiment. Fig. 4 presents the ranking algorithms considering the execution time and it can be also observed that the CCF obtained the best times for both cases of errors occurrence or lack of errors. The closest results were obtained for MCP algorithm, but from the resources load-balancing point of view it offers worst results than CCF algorithm.

For CPU utilization analysis the tests were conducted for both the case with and without errors occurrence (see Fig. 5).

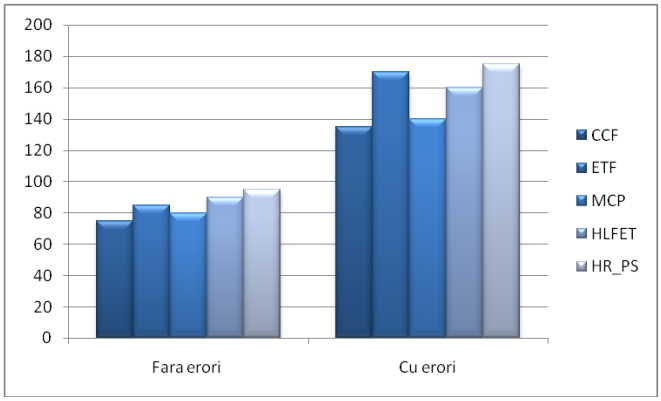

Fig. 4. The comparison between execution time (s) for scheduling algorithms for both cases with and without errors occurrence

The lowest CPU utilization was obtained for the CCF scheduling algorithm, for both cases with and without errors occurrence, which is explained by the higher number of transfers between different processors involved in the simulation experiment. Furthermore CCF offers a better load balancing. An appropriate execution time for scheduled tasks was obtained for MCP scheduling algorithm. Another interesting observation is about ETF algorithm which, in spite of its strategy to keep the CPU as occupied as possible, the results puts it on the last place.

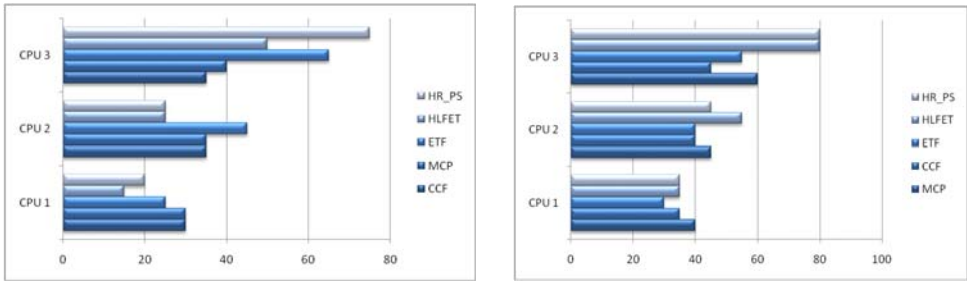

Fig. 5. The CPU Utilization (%) results obtained in each experiment: with and without errors occurrence

Next we present the second tests set results (see Fig. 6). Since previous tests have shown that the CCF algorithm get the best results, it was used for this set of tests. Rescheduling algorithm performance analysis for large graphs (25, 50, 100, 500 nodes) obtained using a graph generator is desired. For a pertinent analysis of the obtained results, with the increasing number of nodes in the graph

we have restricted the percentage range for the submitted tasks number. This is highlighted in Fig. 6.

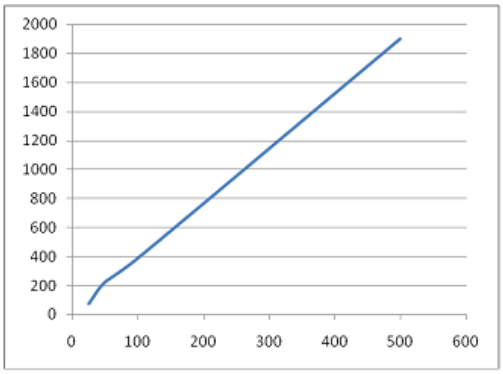

Fig. 6. The variation of the submitted tasks number according to the initial number of tasks sent to the system

7. Conclusions and future work

The rescheduling algorithm proposed in this paper has an important feature: is a generic algorithm, because it can be used with a large variety of rescheduling heuristics. It may also use more rescheduling strategies whose classification was elaborated according to the node position in graphic and depending on the fault tolerance mechanisms that can be used.

Choosing a good scheduling approach is also an important issue for the performance of an application launched onto a distributed systems environment. Many scheduling algorithms have been proposed, studied and compared, but there are few studies comparing the performance of scheduling algorithms considering at the same time the distributed system structure on which we want to schedule the tasks, the type of directed acyclic graph (DAG), in which graph nodes represent tasks and graph edges represent data transfers, and the type of tasks, for example CPU-bound vs. I/O bound. Choosing the best known scheduling algorithm can improve performance of an application if all the aspects previous enumerated are considered. Our future work is aimed to propose a method for choosing, in a dynamic manner, the most appropriate scheduling algorithm for a particular distributed system, which is analyzed.

Acknowledgments

The research presented in this paper is supported by national project, *SORMSYS - RESOURCE MANAGEMENT OPTIMIZATION IN SELF-ORGANIZING LARGE SCALE DISTRIBUTED SYSTEMS*, Project CNCSIS-PN-II-RU-PD ID: 201 (Contract No. 5/28.07.2010).

REFERENCES

[1] *S.A. de Sousa, F.J. da Silva e Silva, R.F. Lopes*, A Flexible Fault-Tolerance Mechanism for the Integrade Grid Middleware, Networking and Services, International conference on, p. 26, International Conference on Networking and Services (ICNS '07), 2007

[2] *R. Sakellariou, H. Zhao,* A low-cost rescheduling policy for efficient mapping of workflows on grid systems, Sci. Program. 12, 4 (Dec. 2004), 253-262, 2004

[3] *A. Forti,* DAG Scheduling for Grid Computing systems, PhD thesis, University of Udine - Italy, 2006

[4] *B. Nazir, K. Qureshi, P. Manuel,* Adaptive checkpointing strategy to tolerate faults in economy based grid, J. Supercomput. 50, 1 (Oct. 2009), 1-18.

[5] *Gabriel Antoniu, Jean-Francois Deverge, Sebastien Monnet,* Building fault-tolerant consistency protocols for an adaptive grid data-sharing service, Project PARIS, September 2004

[6] *G.V. Iordache, M.S. Boboila, F. Pop, C. Stratan, V. Cristea,* A decentralized strategy for genetic scheduling in heterogeneous environments, Multiagent Grid Syst. 3, 4 (Dec. 2007), 355-367

[7] *I. Hernandez, M. Cole,* Reliable DAG scheduling on grids with rewinding and migration, Proceedings of the First international Conference on Networks For Grid Applications (Lyon, France, October 17 - 19, 2007). ICST (Institute for Computer Sciences Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, 1-8

[8] *C. Dobre, C. Stratan, V. Cristea,* Realistic Simulation of Large Scale Distributed Systems using Monitoring, Proceedings of the 2008 international Symposium on Parallel and Distributed Computing (July 01 - 05, 2008), ISPDC. IEEE Computer Society, Washington, DC, 434-438.