A Fault Tolerant Scheduling Algorithm for DAG Applications in Cluster Environments

Nabil Tabbaa, Reza Entezari-Maleki, and Ali Movaghar

Department of Computer Engineering, Sharif University of Technology,
Tehran, Iran
{tabbaa,entezari}@ce.sharif.edu, movaghar@sharif.edu

Abstract. Fault tolerance is an essential requirement in systems running applications which need a technique to continue execution where some system components are subject to failure. In this paper, a fault tolerant task scheduling algorithm is proposed for mapping task graphs to heterogeneous processing nodes in cluster computing systems. The starting point of the algorithm is a DAG representing an application with information about the tasks. This information consists of the execution time of the tasks on the target system processors, communication times between the tasks having data dependencies, and the number of the processor failures (ε) which should be tolerated by the scheduling algorithm. The algorithm is based on the active replication scheme, and it schedules ε+1 replicas of each task to achieve the required fault tolerance. Simulation results show the efficiency of the proposed algorithm in spite of its lower complexity.

Keywords: Cluster Environment, Task Scheduling Algorithms, DAG Tasks, Fault Tolerance.

1 Introduction

Cluster environments consist of an array of diverse computers connected by high-speed networks to achieve powerful platforms. Cluster computing systems are widely deployed for executing computationally intensive parallel applications with various computing requirements [1]. Although the field of parallel computing has existed for many years, programming a parallel system to execute a single application is still a challenging problem, strongly more challenging than programming a single processor, or a sequential system. Allocation of the tasks to the processors and specifying the order of the execution is one of the most important steps in parallel programming. This step, named scheduling, fundamentally determines the efficiency of the application's parallelization. The parallelization in parallel programming shows the speedup of the execution in comparison to a single processor system [2].

There are two well-known types of scheduling algorithms; dynamic and static scheduling. In dynamic scheduling, the decision as to which processor executes a task and when is controlled by the runtime system. This is mostly practical for independent tasks. In contrast, static scheduling means that the processor allocation, often called mapping, and the ordering of the tasks are determined at compile time.

V. Snasel, J. Platos, and E. El-Qawasmeh (Eds.): ICDIPC 2011, Part I, CCIS 188, pp. 189–199, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The advantage of static scheduling is that it can include the dependences and communications among the tasks in its scheduling decisions. Furthermore, since the scheduling is done at compile time, the execution is not burdened with the scheduling overhead [3]. The main goal of most scheduling strategies is to minimize the scheduling length, which is the total completion time of the application tasks. An alternative designation for schedule length, which is quite common in the literature, is makespan [4] and [5].

Resource failures may frequently occur in distributed computing systems and have undesired effects on applications. Consequently, there is an increasing need for developing techniques to achieve fault tolerance [6] and [7]. Fault tolerance is an important property in distributed computing as the dependability of individual resources may not be guaranteed. A fault tolerant approach may therefore be useful in order to potentially prevent a malicious node affecting the overall performance of the application. This subject is very important in distributed computing systems because the size and complexity of the applications are increased dramatically to take advantage of such system resources. Actually, the probability of error occurrence may be increased by the fact that many cluster applications will perform long tasks that may require several days of computation. Hence, the cost and difficulty of recovering from faults in distributed applications are higher than those of traditional applications [6]. If fault tolerance is not provided, the system cannot survive to continue when one or several processors fail. In such situation, the entire program crashes. Therefore, a technique is needed to enable a system to execute critical applications even in the presence of one or more processor failures. Both the task scheduling and fault tolerance within distributed systems are difficult problems in their own, and solving them together makes the problem even harder. Concretely, the main goal of fault tolerant task scheduling algorithms is to find a static schedule of application tasks on the processing elements of a cluster computing system and tolerate a given number of processor failures. The input of the fault tolerant scheduling algorithm is a specification of the application tasks, the computation power of the processing elements, and some information about the execution times of the tasks on the system processors and the communication times between the tasks. In this paper, a fault tolerant task scheduling algorithm is proposed, which aims at tolerating multiple processor failures and tries to achieve a minimum possible makespan. The proposed algorithm uses active replication scheme to mask failures.

The remainder of this paper is organized as follows. Section 2 presents a review of the related works. A brief description of the task graph and the multiprocessor models is given in Section 3. Section 4 presents the proposed algorithm. The simulation results are presented in Section 5. Finally, Section 6 concludes the paper and presents future work.

2 Related Works

A large number of task scheduling algorithms for DAG applications have been proposed in the literature. But most of the available algorithms assume that the processors of the system are completely safe, so they do not tolerate any failure in the system components. Fault tolerance can be achieved in distributed computing systems

by scheduling multiple copies of each task on different processors. In the follow, a brief survey of two well-known types of fault tolerant task scheduling algorithms named primary/backup scheduling and active replication scheduling are presented.

Oh et al. [7] have proposed an algorithm in which each of the submitted tasks are assumed to be independent and non-preemptive. The algorithm considers the case where the backup copies are allowed to be overlapped in time of their execution on a processor; if the primary copies are scheduled on different processors. Ghosh et al. [8] present techniques to provide fault tolerance for non-preemptive, aperiodic and real-time tasks having deadline. The goal of the presented techniques is to achieve high acceptance ratio, percentage of accepted arriving tasks. Manimaran et al. [9] have presented an algorithm to dynamically schedule real-time tasks. This algorithm handles resource constraints, where a task might need some resources, such as data structures, variables, and communication buffers for its execution. Al-Omari et al. [10] have proposed an algorithm which uses the Primary-Backup (PB) overloading technique to be as an alternative to the usually used Backup-Backup overloading. The algorithm is presented to improve schedulability and achieve fault tolerant scheduling of real-time tasks in multiprocessor systems. Zheng et al. [11] and [12] have proposed two techniques, called the Minimum Replication Cost with Early Completion Time (MRC-ECT) and the Minimum Completion Time with Less Replication Cost (MCT-LRC), to schedule backups of independent and dependent jobs, respectively.

The main disadvantage of all of the previous algorithms is that only two copies of the task are scheduled on different processors. Based on this assumption, the task can be completed only when one processor fails. So, these algorithms cannot tolerate more than one failure at a time.

In active replication scheme, multiple copies of each task are mapped on different processors, which are run in parallel to tolerate a given number of failures. Hashimito et al. [13] have proposed a new approach to achieve fault tolerance by scheduling DAG applications on identical processing elements. This algorithm exploits implicit redundancy, which is originally introduced by task duplication to reduce the execution times of parallel programs. Girault et al. [14] have presented an algorithm with the goal of automatically obtain a distributed and fault tolerant task scheduling in embedded systems. The proposed algorithm considers timing constraints on tasks execution, and indicates whether or not the real-time constraints are satisfied. In order to tolerate N failures, the algorithm allows at least N+1 replicas of a task to be scheduled on different processors.

3 The Directed Acyclic Graph Scheduling Problem

The objective of Directed Acyclic Graph (DAG) scheduling is to minimize the overall program finish-time by proper allocation of the tasks to the processors and arrangement of execution sequence of the tasks. Scheduling is done in such a manner that the precedence constraints among the program components are preserved.

3.1 The DAG Model

A parallel program can be represented by DAG, $G = (V, E)$, where V is a set of v nodes and E is a set of e directed edges. Each node n_i in the DAG denotes a task, and

its weight represents the computation cost and is indicated by $w(n_i)$. The edges in the DAG, each of which is denoted by (n_i, n_j), correspond to the communication messages and precedence constraints between the nodes. The weight of an edge is called the communication cost and is indicated by $c(n_i, n_j)$. The communication has no cost if two nodes are mapped to the same processor. For a node n_i in G, $pred(n_i)$ is the set of immediate predecessors and $succ(n_i)$ denotes its immediate successors. A node having no parent is called an entry node and a node having no child is called an exit node [3]. The precedence constraints of a DAG dictate that a node cannot start execution before it gathers all of the messages from its parent nodes. A critical path (*CP*) of a DAG is a longest path traversed from an entry node to an exit node. Obviously, a DAG can have more than one *CP*. Consider the task graph shown in Fig. 1. In this task graph, nodes n_1, n_7, and n_9 are the nodes of the only *CP*. The edges on the *CP* are shown with thick arrows. The communication-to-computation-ratio (*CCR*) of a parallel program is defined as its average edge weight divided by its average node weight [15]. Hereafter, the terms node and task are used interchangeably.

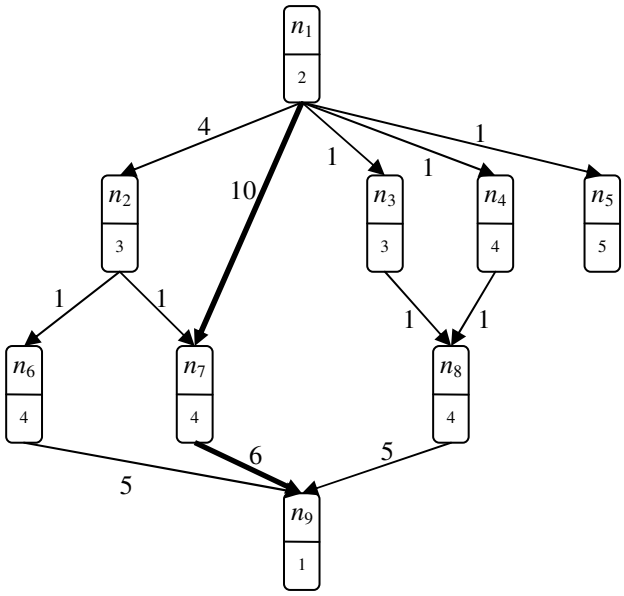

Fig. 1. Directed Acyclic Graph

3.2 The Multiprocessor Model

In DAG scheduling, the target system is represented by a finite processor set $P = \{P_1, P_2, \ldots, P_m\}$. The processors may be heterogeneous or homogeneous. The heterogeneity of the processors means that they have different speeds or processing capabilities. However, it is assumed that every task of the application can be executed on any processor even though the completion times on different processors may be different. The heterogeneity of processing capability is modeled by a function

$C:P \rightarrow R^+$, so the completion time of task n_i on processor P_k equals to $C(P_k) \times w(n_i)$ [15]. The processors are assumed to be fully connected.

4 The Proposed Algorithm

The objective of the proposed algorithm is to map the tasks of DAG represented application to processors with diverse capabilities in a cluster computing system. The algorithm aims to minimize the schedule length while tolerating a given number of ε fail-silent (fail-stop) processor failures. To achieve this, active replication scheme is used to allocate ε +1 copies of each task to different processors.

4.1 Scheduling Heuristic

The proposed algorithm mainly uses the well-known heuristic technique encountered in scheduling algorithms that is called list scheduling. In its general form, the first part of list scheduling sorts the nodes of the application graph to be scheduled depending on a priority scheme, while respecting the precedence constraints of the nodes. In the second part, each node of the list is consecutively scheduled to a processor chosen for the node [3]. In our algorithm each node is scheduled to multiple processors to achieve the required fault tolerance. An important characteristic of list scheduling is that it guarantees the feasibility of all partial schedules, and the final schedule, by scheduling only free nodes and choosing an appropriate start time for each node [3]. The nodes are processed in precedence order (i.e., in topological order), so at the time a node is scheduled all ancestor nodes have already been processed.

4.2 Priority Scheme

List scheduling algorithms establish the scheduling order of the nodes before the scheduling process. During the node scheduling in the second part, this order remains unchanged, so the node priorities are static. To achieve most efficient schedules, it is better to consider the state of the partial schedule when the order of the remaining nodes is established. In this case, the priorities of the nodes are considered to be dynamic. Additionally, the node order must be compatible with the precedence constraints of the application graph, which is achieved if only free nodes are scheduled.

In the proposed algorithm free nodes are ordered by a priority value equals to *tlevel*+*blevel* of the node, where *tlevel* and *blevel* denote the *dynamic top level* and the *static bottom level* of the node respectively. The word *dynamic* implies that the value *tlevel* depends upon the nodes which have already been mapped, and the word *static* implies that the value *blevel* remains unchanged during the scheduling process.

Taking the computational heterogeneity of the system into account, the average execution time of a node on all processors can be used when calculating *blevel*, since the processor on which a node will be assigned is not known. So *blevel* can be computed using (1).

$$blevel(n_i) = \max_{n_j \in succ(n_i)} \{\overline{w(n_i)} + c(n_i, n_j) + blevel(n_j)\} \ . \quad (1)$$

Where $\overline{w(n_i)}$ is the average execution time of node n_i and $c(n_i, n_j)$ is the communication cost between node n_i and node n_j (a successor of n_i).

The *tlevel* is calculated dynamically for each of the free nodes at each step by (2).

$$tlevel(n_i) = \max_{n_j \in pred(n_i)} \{FT(n_j, \Pr oc(n_j)) + c(n_i, n_j)\} \ . \quad (2)$$

Where $FT(n_j, \text{Proc}(n_j))$ is the finish time of node n_j (a predecessor of n_i) which has been previously scheduled on processor $\text{Proc}(n_j)$.

This priority value provides a good measure of the node importance, since the nodes that have the maximum value of *tlevel*+*blevel* compose the critical path of the application graph. The greater the priority, the more work is to be performed along the path containing that node.

4.3 Processor Choice

At each step the scheduling process selects the free node n that has the highest priority and tries to schedule it on all processors to calculate its expected finish time on each processor using (3)

$$FT(n, P_l) = w(n, P_l) + \max\{\max_{n_j \in pred(n)} [\min_{1 \le k \le \varepsilon+1} (FT(n_j^k, \Pr oc(n_j^k)) + c(n_j, n), r(P_l)]\} \cdot \quad (3)$$

Where $r(P_l)$ is the ready time of the processor P_l, and the predecessor nodes are already scheduled onto $\varepsilon + 1$ processors, and n_j^k denotes the replicas of node n_j.

Then, the node n is scheduled on the $\varepsilon + 1$ processors which deliver the minimum finish time for that node using (3). Actually, (3) determines the finish time of the node n if no processor fails during the execution of the application, since the minimum of all replicas is used. In this case, the lower bound of the schedule length $SL_{\min}$ can be computed using (4).

$$SL_{\min} = \max_{n \in V} \{\min_{1 \le k \le \varepsilon+1} [FT(n^k, \Pr oc(n^k))]\} \ . \quad (4)$$

While for the worst case, in the presence of ε failures, the finish time would be given by (5).

$$FT(n, P_l) = w(n, P_l) + \max\{\max_{n_j \in pred(n)} [\max_{1 \le k \le \varepsilon+1} (FT(n_j^k, \Pr oc(n_j^k)) + c(n_j, n), r(P_l)]\} \cdot \quad (5)$$

Then, to compute the upper bound of the schedule length $SL_{\max}$, (6) can be used.

$$SL_{\max} = \max_{n \in V} \{\max_{1 \le k \le \varepsilon+1} [FT(n^k, \Pr oc(n^k))]\} \ . \quad (6)$$

4.4 The Algorithm

The main steps of the proposed scheduling algorithm can be written as follows:

1) Compute *blevel* for each task in the graph,
2) Mark all entry tasks as free tasks,
3) **While** still have unscheduled tasks do
4) Compute *tlevel* for each free task,
5) Update the priorities of all free tasks,
6) Select a free task n with highest priority,
7) Compute the finish time $FT(n, P_l)$ of the task n on all of the processors,
8) Schedule the task n on $\varepsilon + 1$ processors that allow the minimum finish time,
9) Add free successors of n to the free tasks,
10) **End while**

5 Simulation Results

To evaluate the proposed fault tolerant scheduling algorithm, this algorithm is simulated and compared to the FTBAR algorithm [14] which is the closest to our algorithm found in the literature. The goal of our simulations is to evaluate the fault tolerance overhead of the proposed algorithm and compare it with the overhead of FTBAR algorithm.

The proposed algorithm and FTBAR are simulated with a set of randomly generated graphs. Different methods of generating random DAGs for simulation can be found in [16]. In this paper, the method of Layer-by-Layer is used in simulation phase. A random graph is generated as follows: given the total number of tasks, we randomly generated a set of levels with a random number of tasks such that the sum of the number of tasks in all of the levels is equal to the total number of tasks. Consequently, the tasks at a given level are randomly connected to the tasks at higher levels. The execution times of the tasks and communication times between them are randomly selected from uniform distributions with chosen ranges. The number of processors is set to 10 and each point in the shown figures in this paper represents an average over 60 random graphs. The most important metric of the performance of the algorithm is the fault tolerance overhead caused by the active replication scheme. The overhead is computed using the following formula.

$$overhead = \frac{FTSL - nonFTSL}{FTSL} \times 100 \ . \tag{7}$$

Where *FTSL* is the fault tolerant schedule length and the *nonFTSL* is the schedule length produced when the number of failures ε is set to zero.

The average fault tolerance overhead is plotted in Fig. 2 as a function of the number of tasks which is varied uniformly in the range [20, 200]. The communication to computation ratio (*CCR*) is set to 1 and the number of failures ε is set to 2 and 5. This figure shows that the average overhead increases with the number of tasks. This is due to the replication of all tasks and communications.

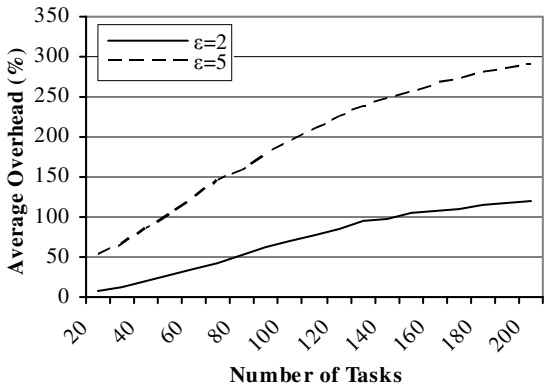

Fig. 2. Average overhead for *CCR*=1

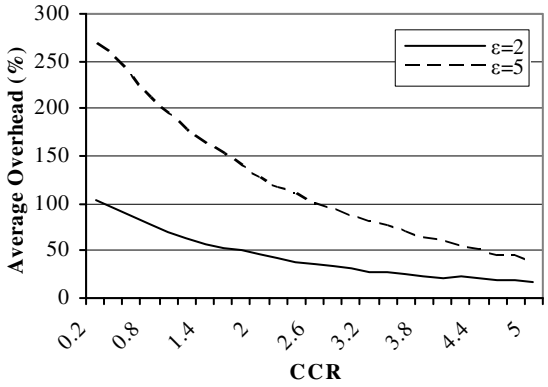

Fig. 3. Average overhead for *Number of Tasks*=100

In Fig. 3 the average fault tolerance overhead is plotted as a function of the *CCR* which is varied uniformly in the range [0.2, 5]. The number of tasks is set to 100 and the number of failures ε is set to 2 and 5. One can see in this figure that the average overhead decreases when the *CCR* increases, since the replication of tasks has a positive effect in withdrawing many of the communications required among the tasks.

Fig. 4 and Fig. 5 show the comparison of the overhead between the proposed algorithm and the FTBAR algorithm as a function of the number of tasks which is varied uniformly in the range [20, 200]. The *CCR* is set to 1 and the number of failures ε is set to 2 and 5 in Fig. 4 and Fig. 5, respectively. It can be seen that the proposed algorithm shows better results compared to FTBAR algorithm for any number of tasks.

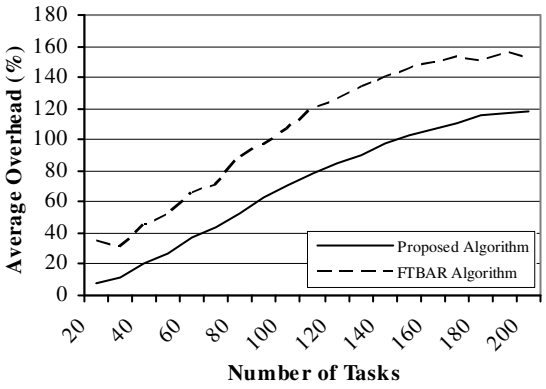

Fig. 4. Average overhead for *CCR*=1 and ε =2

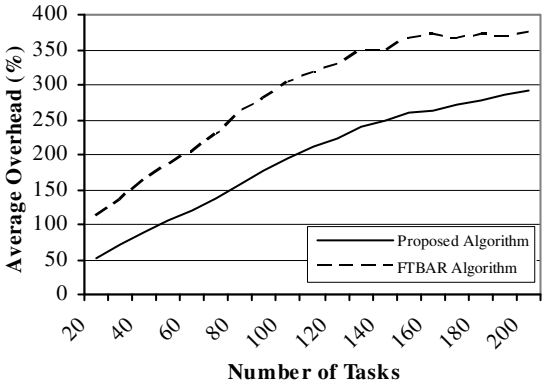

Fig. 5. Average overhead for *CCR*=1 and ε =5

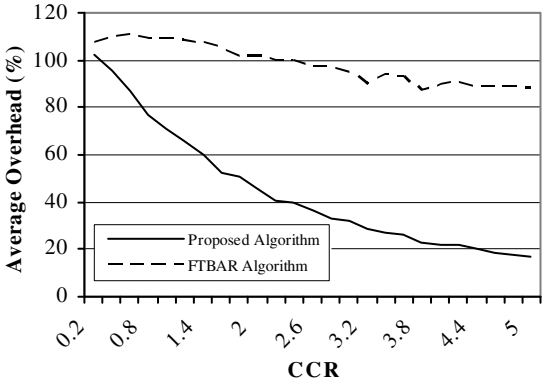

Fig. 6. Average overhead for *Number of Tasks*=100 and ε =2

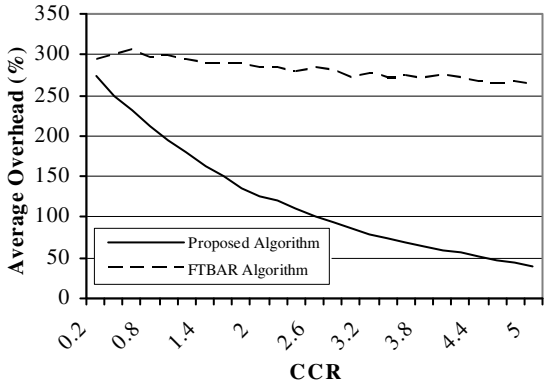

Fig. 7. Average overhead for *Number of Tasks*=100 and ε =5

In Fig. 6 and Fig. 7 the comparison of the overhead is shown as a function of the *CCR* which is varied uniformly in the range [0.2, 5]. The number of tasks is set to 100 and the number of failures ε is set to 2 and 5 in Fig. 6 and Fig. 7, respectively. We can see that for small values of *CCR* there is a little difference between the proposed algorithm and the FTBAR. But for higher values of *CCR*, the proposed algorithm performs significantly better than FTBAR algorithm.

6 Conclusions and Future Work

A large number of algorithms for scheduling and partitioning DAGs have been proposed in the literature, either with an unbounded or with a limited number of processors. Most of these algorithms assume that the processors in the systems are completely safe, so they do not achieve fault tolerance. Some techniques for supporting fault tolerant systems have been proposed, but only few of them are able to tolerate multiple failures at a time. In this paper, a fault tolerant task scheduling algorithm is proposed for mapping DAG tasks on cluster computing systems with heterogeneous processor capabilities. The algorithm is based on active replication, and it schedules ε+1 replicas of each task on different processors to tolerate a given number ε of processor failures. Despite its lower complexity, simulation results demonstrate that the proposed algorithm has an efficient performance in the term of schedule length overhead. It outperforms the closest available algorithm FTBAR, especially in the case of high communication to computation ratio.

Scheduling ε+1 replicas of each task on different processors results in replicating the communications between tasks $(\varepsilon+1)^2$ times. This is due to the fact that each of ε+1 replicas of each task will receive the same message from the ε+1 replicas of each one of its predecessors. Future work on this algorithm might try to reduce the total number of communications. Additionally, in the proposed algorithm, the processors are considered fully connected with non-faulty links. While this can be appropriate in cluster environments, extensions might be added to this algorithm to take communication link failures into account, and make it relevant to other distributed computing systems such as grid environments.

References

1. Buyya, R.: High Performance Cluster Computing: Architectures and Systems, 1st edn. Prentice Hall PTR, Upper Saddle River (1999)
2. Buyya, R.: High Performance Cluster Computing: Programming and Applications, 1st edn. Prentice Hall PTR, Upper Saddle River (1999)
3. Sinnen, O.: Task Scheduling for Parallel Systems, 1st edn. John Wiley and Sons Inc, New Jersey (2007)
4. Entezari-Maleki, R., Movaghar, A.: A genetic-based scheduling algorithm to minimize the makespan of the grid applications. In: Kim, T., Yau, S., Gervasi, O., Kang, B., Stoica, A. (eds.) Grid and Distributed Computing, Control and Automation. Communications in Computer and Information Science, vol. 121, pp. 22–31. Springer, Heidelberg (2010)
5. Parsa, S., Entezari-Maleki, R.: RASA: A new grid task scheduling algorithm. International Journal of Digital Content Technology and its Applications 3(4), 91–99 (2009)
6. Sathya, S.S., Babu, K.S.: Survey of fault tolerant techniques for grid. Computer Science Review 4(2), 101–120 (2010)
7. Oh, Y., Son, S.H.: Scheduling real-time tasks for dependability. Journal of Operational Research Society 48(6), 629–639 (1997)
8. Ghosh, S., Melhem, R., Mosse, D.: Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. IEEE Transactions on Parallel and Distributed Systems 8(3), 272–284 (1997)
9. Manimaran, G., Murthy, C.S.R.: A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. IEEE Transactions on Parallel and Distributed Systems 9(11), 1137–1152 (1998)
10. Al-Omari, R., Somani, A., Manimaran, G.: A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems. In: The 15th International Parallel and Distributed Processing Symposium, pp. 32–39 (2001)
11. Zheng, Q., Veeravalli, B., Tham, C.K.: Fault-tolerant scheduling of independent tasks in computational grid. In: The 10th IEEE International Conference on Communications Systems, pp. 1–5 (2006)
12. Zheng, Q., Veeravalli, B., Tham, C.K.: On the design of fault-tolerant scheduling strategies using primary-backup approach for computational grids with low replication costs. IEEE Transactions on Computers 58(3), 380–393 (2009)
13. Hashimito, K., Tsuchiya, T., Kikuno, T.: A new approach to realizing fault-tolerant multiprocessor scheduling by exploiting implicit redundancy. In: The 27th International Symposium on Fault-Tolerant Computing, pp. 174–183 (1997)
14. Girault, A., Kalla, H., Sighireanu, M., Sore, Y.: An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In: International Conference on Dependable Systems and Networks, pp. 159–168 (2003)
15. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999)
16. Cordeiro, D., Mouni, G., Perarnau, S., Trystram, D., Vincent, J.M., Wagner, F.: Random graph generation for scheduling simulations. In: The 3rd International ICST Conference on Simulation Tools and Techniques, pp. 60:1-60:10 (2010)