Genetic Algorithm for DAG Scheduling in Grid Environments

Florin Pop, Ciprian Dobre, Valentin Cristea
Faculty of Automatic Control and Computer Science, University "'Politehnica"' of Bucharest, Romania
Emails: {florin.pop, ciprian.dobre, valentin.cristea}@cs.pub.ro

***Abstract*—Complex applications are describing using workflows. Execution of these workflows in Grid environments require optimized assignment of tasks on available resources according with different constrains. This paper presents a decentralized scheduling algorithm based on genetic algorithms for the problem of DAG scheduling. The genetic algorithm presents a powerful method for optimization and could consider multiple criteria in optimization process. Also, we describe in this paper the integration platform for the proposed algorithm in Grid systems. We make a comparative evaluation with other existing DAG scheduling solution: Cluster ready Children First, Earliest Time First, Highest Level First with Estimated Times, Improved Critical Path with Descendant Prediction) and Hybrid Remapper. We carry out our experiments using a simulation tool with various scheduling scenarios and with heterogeneous input tasks and computation resources. We present several experimental results that offer a support for near-optimal algorithm selection.**

***Keywords*-Grid Scheduling; Genetic Algorithms; Simulation; Monarc.**

I. Introduction

A Grid is a large scale distributed system with an infrastructure that covers a set of heterogeneous machines located in various organizations and geographic locations. It is basically a collection of computing resources which are used by applications to perform tasks [1]. The Grid architecture must provide good support for resource management as well as adaptability and scalability for applications. The real problem consists in "coordinated resource sharing" (computers, software, data and other resources) and "problem solving in dynamic, multi-institutional virtual organizations" [2], [3]. In most cases, heterogeneous distributed systems have proved to produce higher performance for lower costs than a single high performance computing machine.

Grid users submit complex applications to be executed on available resources provided by the Grid infrastructure, setting a number of restrictions like time (deadline), quality, and cost of the solution. These applications are split into tasks with data dependencies. Two types of workflows can be distinguished: static and dynamic. The description of a static workflow is invariant in time. A dynamic workflow changes during the workflow enactment phase due to circumstances unforeseen at the process definition time [4].

NP-hard problems are often optimized using stochastic search algorithms like simulated annealing or genetic algorithms. Task scheduling is no exception and the genetic algorithm method especially has gained cosiderable popularity. If the tasks are part in a workflow, the scheduling problem becomes more compicated.

We present in this paper a genetic algorithm for DAG scheduling in Grid environments. The paper starts with a background of DAG scheduling problem and presents the goal of DAG scheduling optimization. Section 3 presents the releated work in the filed of DAG scheduling algorithms and offers a critical analisys of existing methods. Section 4 presents the proposed schedulung algorithm and described the used genetic operators. In Section 5 we present the experimental results that validate the proposed algorithm. The paper ends with concluding remarks ans possible future work.

II. Background of DAG scheduling problem

A task graph is a directed acyclic graph $G(V, E, c, \tau)$ where:

- V is a set of nodes (tasks);
- E is a set of directed edge (dependencies);
- $c : V \rightarrow R_+$ is a function that associates a weight $c(u)$ to each node $u \in V$; $c(u)$ represent the execution time of the task T_u, which is represented by the node u in V;
- τ is a function $c : E \rightarrow R_+$ that associates a weight to a directed edge; if u and v are two nodes in V then $\tau(u, v)$ denotes the inter-tasks communication time between T_u and T_v.

A task graph example is shown in Figure 1. Two special kinds of nodes can be identified. A source node is a node without incoming directed edges. It corresponds to the task that initiate the entire application, and it is the first task in any possible schedule. An exit node is a node without outgoing directed edges. In the Figure 1 the node 1 represents the source node and the node 9 represents the exit node. Two items are associated with each node: the task id T_u is represented in the upper half of the node (circle), while the execution time $c(u)$ is represented in the lower half. Each edge is labeled with the inter-tasks communication time.

If we denote $st(u)$ the start time and $ft(u)$ the finish time for task u, and define $makespan = max_{u \in V} \{ft(u)\}$ we can formulate the goal of optimizing DAG scheduling as follows: minimize the $makespan$ without violating precedence constrains.

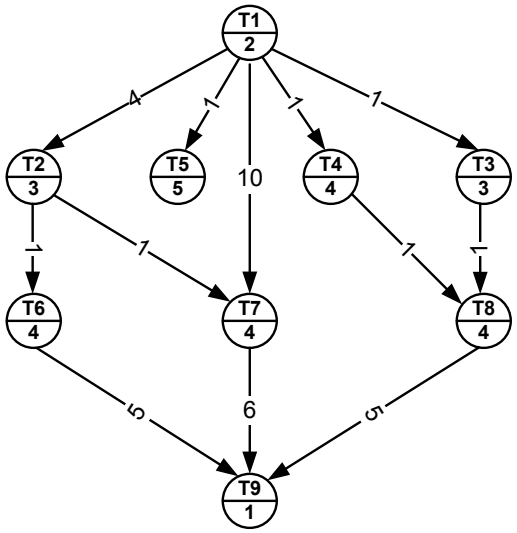

Figure 1. Task DAG Example

A scheduler is considered efficient if the makespan is short and respects resource constrains, such as a limited number of processors, memory capacity, available disk space, etc.

Many types of scheduling algorithms for DAG are based on the list scheduling technique. Each task has an assigned priority, and scheduling is done according to a list priority policy: select the node with the highest priority and assign it to a suitable machine. According to this policy, two attributes are used for assigning priorities:

- *tlevel* (top-level) for a node u is the weight of the longest path from the source node to u.
- *blevel* (bottom-level) for a node u is the weight of the longest path from u to an exit node.

The time-complexity for computing tlevel and blevel is $O(|V| + |E|)$.

We define the $ALAP$ (As Late As Possible) attribute for a node u to measure how far the node's start-time, $st(u)$ can be delayed without increasing the $makespan$. This attribute will have an important role for load balancing constrains because it show if we can delay the execution start of a task T_u. For the example in Figure 1, for the node 1 (task T_1) the $tlevel$ is 0. It means that this node must be scheduled and executed without any time delay. The $blevel$ is 23 and denotes the makspan for this graph. For the node 9 (task T_9) the $ALAP$ is 22 (the maximum time before task 9 ca start the execution).

For DAG scheduling, the goal of optimization is to minimize the total schedule length ($makespan$). The main problem raised by the workflow consists of submitting the scheduled tasks to Grid resources without violating the structure of the original workflow [5].

The critical path (CP) is the weight of the longest path in the DAG and offers an upper limit for the scheduling cost. Algorithms based on "critical path" heuristics produce the best results on average. They take into consideration the critical path of the scheduled nodes at each step. However, these heuristics can result in a local optimum, failing to reach the optimal global solution [6].

The DAG scheduling problem is an NP-complete problem [7], [8]. A solution for this problem consists of a series of heuristics [9], where tasks are assigned priorities and placed in a list ordered by priority. The method through which the tasks are selected to be planned at each step takes into consideration this criterion, thus the task with higher priority receives access to resources before those with a lower priority. The heuristics used vary according to task requirements, structure and complexity of the DAG.

Most scheduling algorithms are based on the so-called list scheduling technique. The basic idea of list scheduling is to make a scheduling list (a sequence of nodes for scheduling) by assigning them some priorities, and then repeatedly execute the following two steps until all the nodes in the graph are scheduled: 1. *Remove the first node from the scheduling list*; 2. *Allocate the node to a processor which allows the earliest start-time*.

If the communication between tasks is considered, there are three models of communication delay:

- *intra-task-communication*: communication delays are implicitly hidden in the topology of the multiprocessor tasks,
- *inter-task-communication*: communication delays occur if dependend uni-proccesor tasks are not processed by the same processor of machine,
- *combination of both*: in the case of divisible task scheduling a multiprocessor task can be partitioned into smaller tasks, one partial task is processed by the current processor and the other parts are distributed among the grid processors, phases of inter-task-communication and computation (with intra-task-communication) alternate with each other.

The *work* done in order to process a task is defined as its running time multiplied by the number of processors assigned to it. Similarly, the work of set of tasks and the work of a schedule are defined:

$$work_i = \tau_i \cdot |proc_i|$$

Usually it is assumed that in the case of malleable tasks the work of a task cannot be decreased by spending more processors on it (preservation of work). Similarly the work of a task cannot be decreased by using virtualization.

Efficiency at time t ($Ef(t)$) is the number of active (busy) processors divided by the total number of processors (active + idle).

$$Ef(t) = \frac{ProcBusy(t)}{ProcActive(t) + ProcIdle(t)}$$

In general we are looking for a feasible solution to scheduling problem. This is a schedule which meets all the

requirements and constrains posed by the problem definition. In addition we may define an objective function that has to be optimized.

III. Critical analisys of existing DAG scheduling algorithms

Priorities of nodes can be determined in many ways such as: HLF (Highest Level First), LP (Longest Path), LPT (Longest Processing Time) or CP (Critical Path). Frequently used attributes for assigning priority include the $t-level$ (top level), the $b-level$ (bottom level), the ALAP (As Late As Possible) and CP (Critical Path) [6]. A taxonomy of task dependency scheduling algorithms in grid environments is presented in Figure 2 [10].

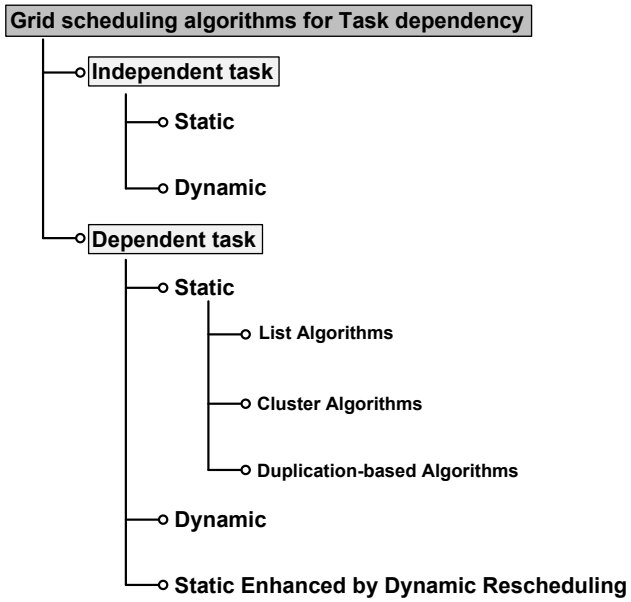

Figure 2. Taxonomy of task dependency scheduling algorithms in Grid environments [10]

Scheduling algorithms for dynamic workflows are based on a dynamic list scheduling. In a traditional scheduling algorithm, the scheduling list is statically constructed before node allocation begins, and most importantly, the sequencing in the list is not modified. In contrast, after each allocation, the dynamic algorithms re-compute the priorities of all unscheduled nodes, which are then used to rearrange the sequencing of the nodes in the list.

Thus, these algorithms essentially employ the following approaches:

1) Determine new priorities of all unscheduled nodes;
2) Select the node with the highest priority for scheduling;
3) Allocate the node to the processor which allows the earliest start-time.

Scheduling algorithms that employ this three-step approach can potentially generate better schedules. However, a dynamic approach can increase the time-complexity of the scheduling algorithm [6].

When tasks are scheduled to resources, there may be some holes between scheduled tasks due to dependences among the tasks of the application. When a task is scheduled to the first available hole, this is called the insertion approach. If the task can only be scheduled after the last task scheduled on the resource without considering holes, it is called a non-insertion approach. The insertion approach performs much better then the non-insertion one, because it utilizes idle times better. However, the complexity of the non-insertion approach is $O(|V| * |P|)$ whereas that of the insertion approach is $O(|V|^2)$, where $|V|$ is the number of nodes and $|P|$ is the number of processing resources.

Different algorithms use the $tlevel$ and $blevel$ in different ways. Some algorithms assign a higher priority to a node with a smaller $tlevel$ while some algorithms assign a higher priority to a node with a larger $blevel$. Still some algorithms assign a higher priority to a node with a larger $(blevel - tlevel)$. In general, scheduling in a descending order of $blevel$ tends to schedule critical path nodes first, while scheduling in an ascending order of $tlevel$ tends to schedule nodes in a topological order.

This section continues with a brief description of a number of static scheduling algorithms. The performance of these algorithms on some primitive graph structures is discussed. A comparison is made among the performance of the scheduling algorithms regarding the total schedule time and the load balancing on the processors. Three of the most popular algorithms tested with very good results are taken into consideration: HLFET, ETF and MCP.

We also presents a new algorithm for DAG scheduling named ICPDP. The result for each algorithm is a total schedule length. The input for these algorithms is presented by example in Figure 1. The comparison between algorithms is represented graphically in the Gantt chart in Figure 3.

The *HLFET* (*Highest Level First with Estimated Times*) algorithm [11] is one of the simplest scheduling algorithms. The algorithm schedules a node to a processor that allows the earliest start time. The main problem with HLFET is that in calculating the SL of a node, it ignores the communication costs on the edges. The HLFET algorithm is a list scheduling algorithm. The time-complexity of the HLFET algorithm is $O(|V|^2)$.

The *ETF* (*Earliest Time First*) algorithm [12] computes, at each step, the earliest start times for all ready nodes and then selects the one with the smallest start time. Here, the earliest start time of a node is computed by examining the start time of the node on all processors exhaustively. When two nodes have the same value of their earliest start times, the algorithm breaks the tie by scheduling the one with the higher SL. Thus, a node with a higher SL does not necessarily get scheduled first because the algorithm gives a higher priority to a node with the earliest start time. The ETF algorithm is

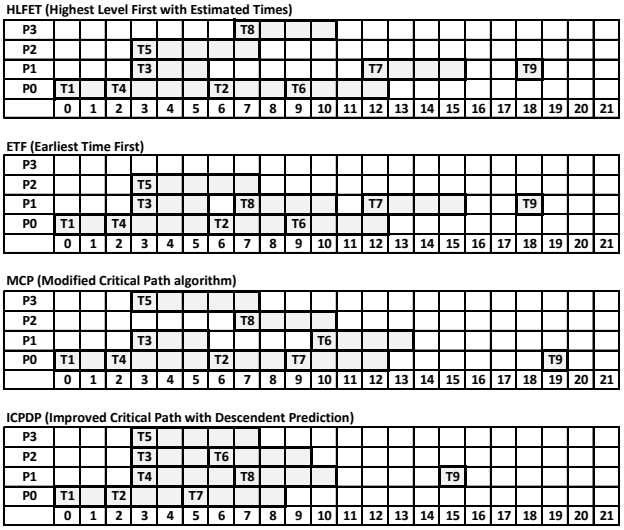

Figure 3. Comparison of tasks allocation for HLFET, ETF, MCP and ICPDP algorithms

also a list scheduling algorithm. The time-complexity of the HLFET algorithm is $O(|P| * |V|^2)$. A total schedule length for ETF is 19 time units, the same as for HLFET.

The *MCP* (*Modified Critical Path*) algorithm [13] uses the $ALAP$ time of a node as a scheduling priority. The $ALAP$ time of a node is computed by first computing the length of CP and then subtracting the $b-level$ of the node from it. Thus, the $ALAP$ times of the nodes on the CP are just their $t-levels$. The MCP algorithm first computes the $ALAP$ times of all the nodes and then constructs a list of nodes in ascending order of $ALAP$ times. Ties are broken by considering the $ALAP$ times of the children of a node. The algorithm then schedules the nodes on the list one by one such that a node is scheduled to a processor that allows the earliest start time using the insertion approach. Basically, the MCP algorithm looks for an idle time slot for a given node. The time-complexity of the MCP algorithm is $O(|V|^2 * \log |V|)$. Total schedule length for MCP is 20 time units.

The *ICPDP* (*Improved Critical Path using Descendant Prediction*). The MCP performs very well in overall situations, offering a good quality with a reasonable low complexity. However, after analyzing the strategy behind the algorithm there is room for improvements. To improve the MCP algorithm, first of all, the original algorithm suffered a slight modification in order to reduce its complexity. Experiments showed that it is not necessary to use all descendants to break ties [14]. Instead, breaking ties by differentiating one level of descendants can produce almost the same result. Secondly, in the second step, the nodes are sorted in an ascending order of $ALAP$ times and ties are broken by the critical child that has the smallest $ALAP$ time. If the children have the same $ALAP$ time, ties are broken randomly. In this case, we must establish a criteria for eliminating the degree of randomness in the algorithm.

In ICPDP, if nodes have the same $ALAP$ the strategy is to take into consideration the static blevel of the nodes. The node with the greatest static level from the nodes with an identic $ALAP$ times is the one scheduled at the current step. The algorithm doesn't take into consideration the following situation. In the second step there are two nodes n_i and n_j with very close $ALAP$ times, node n_i having a smaller $ALAP$ than node n_j. If node n_j has more descendants on the critical path than node n_i than maybe scheduling node n_j first will open a larger part of the DAG by breaking more dependencies "freeing" more descendants and thus giving a better result in the long run.

This idea is implemented in the ICPDP. The improved algorithm (ICPDP) mentions that if $ALAP$ times are very close within a certain threshold, the node with the greatest number of descendants should be considered. The proximity of $ALAP$ times is measured by a threshold T_{limit}. This limit was first considered to be 0.9, meaning that if the $ALAP$ of node n_i is 0.9 times smaller than the $ALAP$ of node n_j and node n_j has more descendants than node n_i, than node n_j is the next scheduled task. The value of the threshold varies according to the number of tasks, the number of dependencies and of course the number of processors available. The time-complexity of the MCP algorithm is $O(|V|^2)$ [15]. Total schedule length for MCP is 16 time units.

When tasks are scheduled to resources, there may be some *holes* between scheduled tasks due to dependences among the tasks of the application. In the above algorithms, a task is scheduled to the first available *hole*, which is called the insertion approach. If the task can only be scheduled after the last task scheduled on the resource without considering *holes*, it is called a non-insertion approach. The insertion approach performs much better then the non-insertion one, because it utilizes idle times better. However, the complexity of the non-insertion approach for N nodes is $O(|V| * |P|)$ whereas that of the insertion approach is $O(|V|^2)$.

IV. Genetic Algorithm for DAG Scheduling

Genetic algorithms (GAs) are a well known optimization heuristic, especially when dealing with combinatorial problems, and particularly with NP-Complete ones, which are not solvable in a polynomial time.

The GA was first introduced by Holland in 1975 and represent a machine learning optimization method based on a metaphor of the evolution process observed in nature. As described by Goldberg [17], in general terms, a genetic algorithm consists of four parts, each of them with a well defined role.

These search procedures are based on the evolution of a population of individuals (a vector of possible solutions). GAs are part of evolutionary supervised learning methods, as

they require an evaluation function provided by the one who creates of the algorithm (instead of reinforcement learning agents for example).

```
1. Generate an initial population.
2. Select pair of individuals based
 on the fitness function.
3. Produce next generation from the
   selected pairs by performing
   random changes on the selected
   parents (by applying pre-selected
   genetic operators).
4. Test for stopping criterion:
     4.1. return the solutions or
     individuals if satisfied, or
     4.2. go to step 2. if not.
```

In Genetic Algorithms, each chromosome (individual in the population) represents a possible solution to a problem. In the case of scheduling, each chromosome represents a schedule of a group (batch) of tasks on a group of processors. A chromosome can be represented as a sequence of individual schedules (one for each processor in the group) separated by a special value. Each individual schedule is a queue of tasks assigned to that processor. In another representation, each gene is a pair of values (T_j, P_i), indicating that task T_j is assigned to processor P_i. The execution order of tasks allocated to the same processor is given by the positions of the corresponding genes in the chromosome. Tasks allocated to different processors can be executed in parallel. A third representation adopts a matrix structure with processors represented on one dimension and queues represented on the second dimension.

In our *chromosome encoding*, we adopted the second representation, in which each gene is a pair of values (T_j, P_i), indicating that task T_j is assigned to processor P_i, where j is the index of the task in the batch of tasks and i is the processor id. This representation has been regarded in literature [13] as efficient and compact, with reduced computational costs (crossover and mutation are easier to implement on this type of representation).

The *initial population* is initialized by randomly placing each task on a processor. To ensure that the search space is thoroughly explored, the chromosomes are created using different random number generators. Various probabilistic distributions are used by each agent for population initialization (Poisson, Normal, Uniform, Laplace). This random generation of chromosomes can lead to configurations that do not correspond to valid schedules (e.g. the tasks' priorities do not correspond to the specification). The problem can be solved by applying corrections to make the generated chromosome compliant with the specification or by including penalties in the fitness function (see later).

All new chromosomes have a certain probability of being affected by *mutation*. The search space expands to the vicinity of the population by randomly altering certain genes. The adaptive operator that we introduced is more flexible. The novelty of our mutation operator is to dynamically adjust the mutation rate, depending on the fitness variation. In our experiments, we modeled a linear increase in mutation rate when the population stagnates, and a decrease towards a predefined threshold when population fitness varies and the search space has moved to a vicinity.

Genetic operators apply to chromosomes that are selected from the actual population and produce a new generation. Our algorithm implements the *roulette wheel selection method*, which proved to work well in similar studies [16].

The fitness (or objective) function measures the quality of each individual in the population according to some criteria. The objective selected for our research was to obtain a well-balanced assignment of tasks to processors. We have considered T_i to be the total number of tasks assigned to processor i for execution and $t_{i,j}$ the running time of task j on processor i. t_i^p is the execution time for previously assigned tasks to processor i, and t_i^c is the processing time for currently assigned tasks.

First factor introduced for fitness computation for a chromosome c is:

$$f_1 = \frac{t_m}{t_M} = \frac{\min_{1 \le i \le |c|}\{t_i\}}{\max_{1 \le j \le |c|}\{t_j\}}, 0 \le f_1 \le 1$$

where $|c|$ represents the number of tasks in the chromosome c. The factor converges to 1 when t_m approaches t_M, and the schedule is perfectly balanced.

The second factor considered for fitness computation is the average utilization of processors:

$$f_2 = \frac{1}{|c|} \sum_{i=1}^{|c|} \frac{t_i}{t_M}, 0 \le f_2 \le 1$$

.

Zomaya [16] pointed out its utility of reducing idle times by keeping processors busy. Division by makespan is pursued in order to map the fitness values to the interval $[0, 1]$. In the ideal case, the total execution times on the processors are equal and equal to makespan, which leads to a value of 1 for average processor utilization.

Another factor considered in our research represents meeting the imposed restrictions. In a realistic scenario, task scheduling must meet both deadline and resource limitations (in terms of memory, cpu power). The fitness factor is subsequently defined as:

$$f_3 = \frac{T_s(c)}{T}, 0 \le f_3 \le 1$$

.

We consider that $T_s(c)$ denotes the number of tasks which satisfy deadline and computation resource requirements for a chromosome c, and T represents the total number of tasks in the current schedule. This factor acts like a contract penalty on the fitness. Its value varies reaching 1 when all

the requirements are satisfied and proportionally decreases with each requirement that is not met.

The last factor, f_4, consider the dependencies between tasks. This factor is directly related to the length of the associated schedule. For a chromosome c we consider that

$$f_4 = \frac{SL(c)}{t_M}$$

where t_M is the maximum schedule length among all chromosomes ($makespan$) and $SL(c)$ denote the schedule length for c.

The fitness function applied in our research consists of the contribution of the factors presented ($0 \leq F(c) \leq 1$):

$$F(c) = f_1 \times f_2 \times f_3 \times f_4$$

$$F(c) = \left(\frac{t_m}{t_M}\right) \times \left(\frac{1}{|c|}\sum_{i=1}^{|c|}\frac{t_i}{t_M}\right) \times \left(\frac{T_s(|c|)}{T}\right) \times \left(\frac{SL(c)}{t_M}\right)$$

An experimental cluster was configured with 11 nodes which represent heterogeneous computing resources with various processing capabilities and initial loads. We carried out our experiments with complex scheduling scenarios and with heterogeneous input tasks and computation resources. The input tasks represent typical CPU-intensive computing programs (we consider a sets of 50, 100, 1000, 3000, 5000, 7000, 9000, 11000, and 13000 tasks). The testing platform for genetic algorithm was an agent based one. This approch allow to consiger cooperation between cluster nodes. Default parameters for the genetic algorithm were established at 0.9 for crossover rate and 0.005 for mutation rate threshold. The values were experimentally determined, in order to widely and thoroughly explore the search space. Our tested analysis led us to improve upon centralized genetic approaches with respect to scalability and robustness.

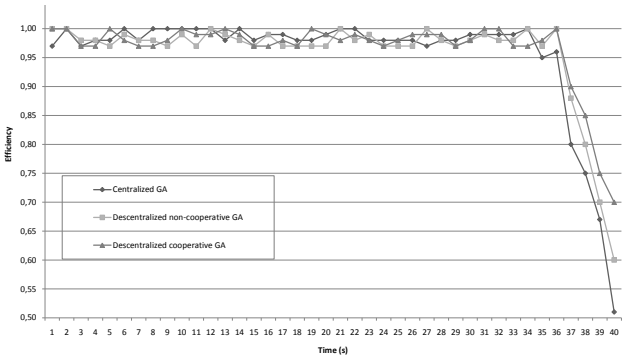

Figure 4. Efficiency of Genetic DAG Scheduling Algorithm

Figure 4 presents the efficiency of genetic algorithm in different experiments. Decentralized Cooperative Genetic Algorithm gives the best results because of the cooperation between agents with a common purpose (the same fitness function), to find a satisfactory near-optimal scheduling solution. The Decentralized Non-Cooperative GA, although benefits from the multiple initial search start points (the results of the algorithm are slightly improved in comparison with those of the Centralized GA), its convergence doesn't improve as fast as in the case of the Cooperative GA because the agents run independently the GA, and are not benefiting from the better solutions found by the others.

V. Conclusions and future work

Scheduling applications on wide-area distributed systems is useful for obtaining quick and reliable results in an efficient manner. Optimized scheduling algorithms are fundamentally important in order to achieve optimized resources utilization. The existing and potential applications include many fields of activity like satellite image processing and medicine.

In this paper it was presented the DAG Grid scheduling issues and the problems of DAG scheduling and the solutions that perform well in most cases. Further, the paper compared clasical DAG scheduling algorithms and ICPDP algorithm, with better performance concerning the schedule length, the total time needed for obtaining the tasks-to-resources mapping solution, a good load balancing and an efficient resource utilization. We used this algorithm to describe the fitnes function in DAG Scheduling algorithm. The testing has been conducted on several test scenarios representing different types of applications with respect to computational cost and communication cost. Scheduling results were gathered from applications with up to 1000 tasks and a high dependencies complexity, in order to offer a good estimation of the genetic algorithm performance. The DAG Scheduling algorithm has demonstrated to offer the best schedule length over all resources in a slightly shorter time than the other scheduling algorithms. Furthermore, the distribution of the tasks to the available resources has proved to accomplish good load balancing and efficient resource allocation by minimizing the idle times on the processing elements.

The future work will considers a large distributes system for test and tunning. The improvment of fitness function with multiple criteria will be another possible direction for algorithm extension.

Acknowledgment

This research is supported by DEPSYS (Models and Techniques for Ensuring Reliability, Safety, Availability and Security of Large Scale Distributes System) project, funded by the CNCSIS in competition PN-II-ID-PCE-2008 (Project ID: 1710).

REFERENCES

[1] Daniel Walton. The simulation of dynamic resource brokering in a grid environment. Technical report, B.Comp.Sc., Department of Computer Science, The University of Adelaide, South Australia, 2002.

[2] I. Foster and C. Kesselman. The grid: Blueprint for a new computing infrastructure. *Morgan Kaufmann*, 1999.

[3] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001.

[4] Fangpeng Dong and Selim Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, School of Computing, Queen's University, Kingston, ON, K7L 3N6, Canada, 2006.

[5] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freud. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, November 1999.

[6] Y.K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), 1999.

[7] M. Kafil and I. Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6(3):42–51, 1998.

[8] J. Ullman. Np-complete scheduling problems. *Computer System Science 10*, 1975.

[9] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling dags on multiprocessors. *Parallel Distributed Computer*, 1992.

[10] F. Pop and V. Cristea. Intelligent strategies for dag scheduling optimization in grid environments. In *Proceedings of the 16th International Conference on Control Systems and Computer Science (CSCS16'07), May 22-25, Bucharest, Romania, Printech, ISBN 978-973-718-743-7*, pages 98–103, 2007.

[11] K.M. Chandy, T.L. Adam, and J. Dickson. A comparison of list scheduling for parallel processing systems. *Communication of ACM*, 17, 1974.

[12] Y. Li and W. Wolf. Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors. *European Design and Test Conference*, 1997.

[13] M.Y. Wu and D.D. Gajski. Hypercool: a programming aid for message-passing systems. *IEEE Transactions Parallel Distributed Systems*, 1990.

[14] M.Y. Wu. Mcp revisited. Department of Electrical and Computer Engineering, The University of New Mexico, 2000.

[15] Bogdan Simion, Catalin Leordeanu, Florin Pop, and Valentin Cristea. A hybrid algorithm for scheduling workflow applications in grid environments (icpdp). In *Proceedings of On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, GADA, and ODBASE and IS, Volume 4804/2007, Vilamoura, Portugal, November 25-30, ISBN: 978-3-540-76835-7, Springer Verlag*, pages 1331–1348, 2007.

[16] A.Y. Zomaya and Y.H. Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):899–911, 2001.

[17] S. Ghosh, R. Melhem, and D. Mosse. 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.