AN ALGORITHM AND SOME NUMERICAL EXPERIMENTS FOR THE SCHEDULING OF TASKS WITH FAULT-TOLERANCY CONSTRAINTS ON HETEROGENEOUS SYSTEMS

Moustafa NAKECHBANDI
Jean-Yves COLIN
LITIS, Le Havre University,
5, rue Philippe Lebon, BP 540, 76058, Le Havre cedex, France.
{moustafa.nakechbandi, jean-yves.colin}@univ-lehavre.fr

KEYWORDS
DAG, Scheduling with communication, Fault tolerant, Heterogeneous systems.

ABSTRACT

In this paper, we propose an efficient scheduling algorithm for problems in which tasks with precedence constraints and communication delays have to be scheduled on an heterogeneous distributed system with an one fault hypothesis. Based on an extension of the Critical-Path Method CPM/PERT, our algorithm combines an optimal schedule with some additional tasks duplication, to provide fault-tolerance. Backup copies are not established for tasks that have already more than one original copy. The result is a schedule in polynomial time that is optimal when there is no failure, and is a good resilient schedule in the case of one server failure. We finally compare the optimal solutions with the resilient solutions found by this algorithm on several semi-random DAGs.

I. INTRODUCTION

Heterogeneous distributed systems have been increasingly used for scientific and commercial applications. Recent examples of such applications include Automated Document Factories (ADF) in banking environments where several hundred thousands documents are produced each day on networks of several multiprocessors servers. Or high performance Data Mining (DM) systems (Palmerini 2004) that need to process very large data collections using very time-consuming algorithms. Or Grid Computing systems (Ruffner et al. 2003, Venugopal et al. 2004) such as Computational Grids which focus primarily on very computationally-intensive operations, or Data Grids which control the sharing and management of large amounts of distributed data.

However, efficiently using these heterogeneous systems is a hard problem, because the general problem of optimally scheduling tasks is NP-complete, even when there are no communication delays (Kwok and Ahmad 1999, Garey and Johnson 1979). When the application tasks can be represented by Directed Acyclic Graphs (DAGs), many dynamic scheduling algorithms have been devised. For some examples, see (Maheswaran and Siegel 1998, Iverson and Özgüner 1998, Chen, and Maheswaran 2002). Also, several static algorithms for scheduling DAGs in meta-computing systems are described in (Colin and Chrétienne 1991, Topcuoglu et al. 1999, Alhusaini, et al. 1999, Kwok and Ahmad 1999). Most of them suppose that tasks compete for limited processor resources, and thus these algorithms are mostly heuristics. Problems with fault tolerant aspects are less studied. Reliable execution of a set of tasks is usually achieved by task duplication and backup copies (Qin and Jiang 2006, Randell 1975, Chen and Avizienis 1978, Girault, et al. 2004).

A very classical and useful tool to study static scheduling problems with DAG is the Critical Path Method (also known as CPM, or PERT method, or CPM/PERT) (Maheswaran and Siegel 1998). Using a relaxation of the constraint on the number of available processors, this method gives results such as a lower bound on the execution time (or makespan) of the application and lower bounds on the execution dates of all tasks of the DAG. Because of the relaxation, tasks can be executed as soon as possible. Improvements and limits of this method to distributed systems with communications delays may be found in (Colin and Chrétienne 1991, Colin et al. 1999, Nakechbandi et al. 2002), for example. In (Colin et al. 2005), we studied the problem of scheduling the tasks of a DAG on the servers of an heterogeneous system. There, the relaxation used in CPM/PERT was replaced by the dual relaxation that each server has no constraint on the number of tasks it can simultaneously process. That is, each server can simultaneously process a non limited number of tasks without loss of performances. Our goal was to compute a lower bound on the execution time of a realistic solution, and compute lower bounds on the execution dates of all tasks of the DAG. In (Nakechbandi et al. 2007), we further supposed that one server (and at most one) could suffer from a crash fault. The algorithm presented there improved on the one presented in (Colin et al. 2005) by adding backup copies to the optimal solution build.

The solution we propose now is simpler than the one presented in (Nakechbandi et al. 2007). Additionally, we present some numerical experiments and simulation results. This rest of this paper is divided into four main parts. In the first one, we present the problem, and in the second one, we present a solution to the problem. In the third part, we make some numerical experiments using randomly generated tasks graphs, comparing the optimal solutions with the resilient solutions found by this algorithm. Finally, in the fourth part, we discuss the advantages and disadvantages of the proposed solution.

II. THE CENTRAL PROBLEM

2.1 The Distributed Servers System

We call Distributed Servers System (DSS) a virtual set of geographically distributed, multi-users, heterogeneous or not, servers. Therefore, a DSS has the following properties: first, the processing time of a task on a DSS may vary from a server to another. The processing time of

Proceedings of the 2008 High Performance Computing & Simulation Conference ©ECMS
Waleed W. Smari (Ed.)
ISBN: 978-0-9553018-7-2 / ISBN: 978-0-9553018-6-5 (CD)

each task on each server is supposed known. Second, although it may be possible that some servers of a DSS are potentially able to execute all the tasks of an application, it may also be possible in some applications that some tasks may not be executed by all servers. In a DSS problem, we suppose that the needs of each task of an application are known, and that at least one server of the DSS may process it.

The classical CPM/PERT relaxation of the number of processors, is replaced in the DSS problem with the dual relaxation that each server has no constraint on the number of tasks it can simultaneously process. Thus we suppose that the concurrent executions of some tasks of the application on a server have a negligible effect on the processing time of any other task of the application on the same server.

The transmission delay of a result between two tasks depends on the tasks and on their respective sites. The communication delay between two tasks executed on the same server is supposed equal to 0.

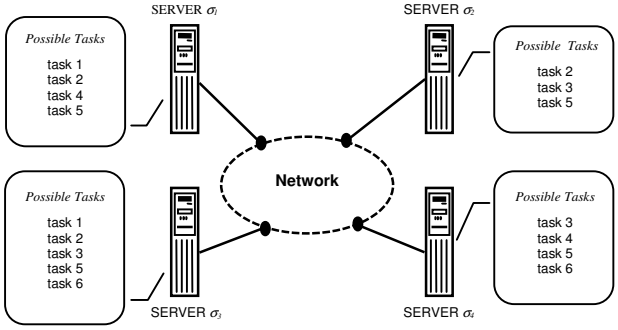

Fig. 1: Example of Distributed Servers System with the list of the executable services for each server.

2.2 Directed Acyclic Graph

An application is decomposed into a set of indivisible tasks that have to be processed. A task may need data or results from other tasks to fulfil its function and then send its results to other tasks. The transfers of data between the tasks introduce dependencies between them. The resulting dependencies form a Directed Acyclic Graph. Because the servers are not necessarily identical, the processing time of a given task can vary from one server to the next. Furthermore, the duration of the transfer of a result on the network cannot be ignored. This communication delay is function of the size of the data to be transferred and of the transmission speed that the network can provide between the involved servers. Note that if two dependent tasks are processed themselves on the same server, this communication delay is considered to be 0.

The central scheduling problem P on a Distributed Server System, is represented therefore by the following parameters:

- a set of servers, noted $\Sigma = \{\sigma_1, ..., \sigma_s\}$, interconnected by a network,
- a set of the tasks of the application, noted $I = \{1,..., n\}$, to be executed on Σ. The execution of task i, $i \in I$, on server σ_r, $\sigma_r \in \Sigma$, is noted i/σ_r. The subset of the servers able to process task i is noted Σ_i, and may be different from Σ,
- the processing times of each task i on a server σ_r is a positive value noted π_{i/σ_r}. The set of processing times of a given task i on all servers of Σ is noted $\Pi_i(\Sigma)$. $\pi_{i/\sigma_r} = \infty$ means that the task i cannot be executed by the server σ_r.
- a set of the transmissions between the tasks of the application, noted U. The transmission of a result of an task i, $i \in$ I, toward a task j, $j \in$ I, is noted (i, j). It is supposed in the following that the tasks are numbered so that if $(i, j) \in$ U, then i < j,
- the communication delays of the transmission of the result (i, j) for a task i processed by server σ_r toward a task j processed by server σ_p is a positive value noted $c_{i/\sigma_r, j/\sigma_p}$. The set of all possible communication delays of the transmission of the result of task i, toward task j is noted $\Delta_{i,j}(\Sigma)$. Note that a zero in $\Delta_{i,j}(\Sigma)$ mean that i and j are on the same server, i.e. $c_{i/\sigma_r, j/\sigma_p} = 0 \Rightarrow \sigma_r = \sigma_p$. And $c_{i/\sigma_r, j/\sigma_p} = \infty$ means that either task i cannot be executed by server σ_r, or task j cannot be executed by server σ_p, or both.

Let $\Pi(\Sigma) = \bigcup_{i \in I} \Pi_i(\Sigma)$ be the set of all processing times of the tasks of P on Σ.

Let $\Delta(\Sigma) = \bigcup_{(i,j) \in U} \Delta_{i,j}(\Sigma)$ be the set of all communication delays of transmissions (i, j) on Σ.

The central scheduling problem P on a distributed servers system DSS can be modelled by a multi-valued DAG $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$. In this case we note $P=\{G, \Sigma\}$.

Example 1 : Figure 2 presents an example of DAG.

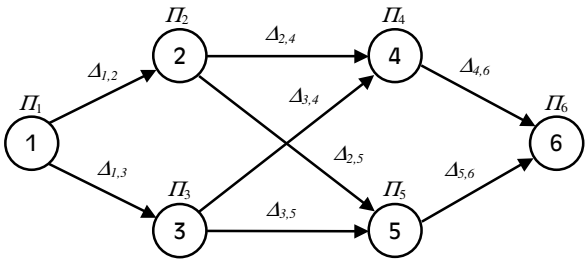

Fig. 2. Example of DAG : the Π_i vector on a node is the vector of the processing time of task i on the various servers, and $\Delta_{i,j}$ on an arc is the communication delays matrix between the two tasks depending on the servers that process them.

On this example, if we have 4 servers $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ and if $\Pi_1 = (\mathbf{3}, \infty, 2, \infty)$, then $\pi_{1/\sigma_1} = 3$. And $\pi_{1/\sigma_2} = \infty$, meaning that server σ_2 cannot execute task 2 etc.

On the same example, communications from task 1 to task 2 are given by matrix $\Delta_{1,2}$ in Fig.3.

	σ_1	σ_2	σ_3	σ_4
σ_1	0	3	2	∞
σ_2	∞	∞	∞	∞
σ_3	2	3	0	∞
σ_4	∞	∞	∞	∞

Fig. 3. Example of communication delays matrix $\Delta_{1,2}$ between task 1 and task 2.

In the matrix of Fig. 3, one can see that if task 1 is processed on server σ_3 and task 2 is processed on server σ_2, then $c_{1/\sigma 3,\, 2/\sigma 2} = 3$.

2.3. Definition of a feasible solution

We note PRED(i), the set of the predecessors of task i in G: $\mathrm{PRED}(i) = \{ k / k \in I \text{ et } (k,i) \in U \}$

And we note SUCC(i), the set of the successors of task i in G: $\mathrm{SUCC}(i) = \{ j / j \in I \text{ et } (i,j) \in U \}$

A feasible solution S for the problem P is a subset of executions $\{ i/\sigma_r , i \in I \}$ with the following properties:

- each task i of the application is executed at least once on at least one server σ_r of Σ_i,
- to each task i of the application executed by a server σ_r of Σ_i, is associated one positive execution date t_{i/σ_r} ,
- for each execution of a task i on a server σ_r, such that PRED(i) ≠ ∅, there is at least an execution of a task k, $k \in$ PRED(i), on a server σ_p, $\sigma_p \in \Sigma_k$, that can transmit its result to server σ_r before the execution date t_{i/σ_r} .

The last condition, also known as the Generalized Precedence Constraint (GPC) (Colin et al. 1999), can be expressed more formally as:

$$\forall i/\sigma_r \in S \begin{cases} t_{i/\sigma_r} \geq 0 & \text{if PRED}(i) = \varnothing \\ \forall k \in \text{PRED}(i), \exists \sigma_p \in \Sigma_k \,/\, t_{i/\sigma_r} \geq t_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r} & \text{else} \end{cases}$$

It means that if a communication must be done between two scheduled tasks, there is at least one execution of the first task on a server with enough delay between the end of this task and the beginning of the second one for the communication to take place. A feasible solution S for the problem P is therefore a set of executions i/σ_r of all i tasks, $i \in I$, scheduled at their dates t_{i/σ_r} , and verifying the Generalised Precedence Constraints GPC. Note that, in a feasible solution, several servers may simultaneously or not execute the same task. This may be useful to generate less communications. All the executed tasks in this feasible solution, however, must respect the Generalized Dependence Constraints.

2.4. Optimality Condition

Let T be the total processing time of an application (also known as the makespan of the application) in a feasible solution S, with T defined as:

$$T = \max_{i/\sigma_r \in S} (t_{i/\sigma_r} + \pi_{i/\sigma_r})$$

A feasible solution S^* of the problem P modelled by a DAG $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$ is optimal if its total processing time T^* is minimal. That is, it does not exist any feasible solution S with a total processing time T such that $T < T^*$.

III. THE DSS_1FAULT ALGORITHM

The algorithm proposed here, named DSS_1FAULT, has two phases: the first one is for the scheduling of original copies where we use the DSS-OPT algorithm (Colin et al. 2005) and the second one is for adding and scheduling additional backups copies when necessary.

3.1. Scheduling the original copies

We schedule original copies of tasks in our algorithm with the DSS-OPT algorithm (Colin et al. 2005). The DSS-OPT algorithm is an extension of CPM/PERT algorithms type to the distributed servers problem. In its first phase, it computes the earliest feasible execution date of each task on every server, and in its second phase it builds a feasible solution (without server fault) starting from the end of the graph with the help of the earliest dates computed in the first phase.

Let P be a DSS scheduling problem, and let $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$ be its DAG.

One can first note that there is an optimal trivial solution to this DSS scheduling problem. In this trivial solution, all possible tasks are executed on all possible servers, as soon as possible, and their results are then broadcasted to all others servers. This is an obvious waste of processing power and communication resources, however, and something as optimal, but less wasteful in terms of used resources, is usually needed.

The first phase of the DSS_OPT routine, DSS_LWB(), goes from the initial tasks to the final ones, computing along the way the earliest feasible execution dates b_{i/σ_r} and earliest end date η_{i/σ_r} , for all possible executions i/σ_r of each task i of problem P.

The second phase of the DSS_OPT routine determines, for every task i that does not have any successor in G, i.e. task i is a "leaf" or final task, the execution i/σ_r ending at the earliest possible date η_{i/σ_r}. If several executions of task i end at the same smallest date b_{i/σ_r} , one is chosen, arbitrarily or using other criteria of convenience, and kept in the solution. Then, for each kept execution i/σ_r that has at least one predecessor in the application, the subset L_i of the executions of its predecessors that satisfy GPC(i/σ_r) is established. This subset of executions of predecessors of i contains at least an execution of each of its predecessors in G. One execution k/σ_p of every predecessor task k of task i is chosen in the subset, arbitrarily or using other criteria of convenience, and kept in the solution. It is executed at its earliest possible date b_{k/σ_p} . The examination of the predecessors is pursued in a recursive manner until the studied tasks do not present any predecessors in G.

3.2. Adding backup copies

The ADD_BACKUP_COPIES routine starts from tasks without any predecessors, similarly to DSS_LWB(), and proceed from there to the end of the DAG. First, if there is currently only one copy of a given task, it determines what is the worst possible delay it may encounter if a failure occurs on another server, while satisfying its GPC. It also determines the fastest server (not considering the server executing the only current copy of this task in the current solution) able to execute this task, and adds a backup copy on this server to the solution, again considering the worst possible delay resulting from this failure, while satisfying the GPC of this copy. Else the task has already several

copies in the optimal solution, and the routine determines for each original copy of this task, what is the worst possible delay it may encounter if a failure occurs on another server, while satisfying its GPC.

The complete DSS_1FAULT algorithm is the following:

Input: $G = \{I, U, \Pi(\Sigma), \Delta(\Sigma)\}$
Output: A feasible solution with backup copies
DSS_1FAULT ()
DSS_OPT() // first phase
ADD_BACKUP_COPIES() // second phase
end DSS_1FAULT
DSS_OPT()
DSS_LWB ()
$T = \max_{\forall i / \mathrm{SUCC}(i)=\emptyset} \min_{\forall \sigma_r \in \Sigma_i} (r_{i/\sigma_r})$
for all tasks i such that SUCC(i) = ∅ **do**
$L_i \leftarrow \{ i/\sigma_r / \sigma_r \in \Sigma_i$ and $r_{i/\sigma_r} \leq T \}$
$i/\sigma_r \leftarrow$ keepOnefrom(L_i)
schedule (i/σ_r)
end for
end DSS_OPT
DSS_LWB()
for each task i where PRED(i) = ∅ **do**
for each server σ_r such that $\sigma_r \in \Sigma_i$ **do**
$b_{i/\sigma_r} \leftarrow 0$
$r_{i/\sigma_r} \leftarrow \pi_{i/\sigma_r}$
end for
mark (i)
end for
while there is a non marked task i such that all its predecessors k in G are marked **do**
for each server σ_r such that $\sigma_r \in \Sigma_i$ **do**
$b_{i/\sigma_r} \leftarrow \max_{\forall k \in \mathrm{PRED}(i)} (\min_{\forall \sigma_p \in \Sigma_k} (b_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r}))$
$r_{i/\sigma_r} \leftarrow b_{i/\sigma_r} + \pi_{i/\sigma_r}$
end for
mark (i)
end while
end DSS_LWB
schedule(i/σ_r)
execute the task i at the date b_{i/σ_r} on the server σ_r
if PRED(i) ≠ ∅ **then**
for each task k such that $k \in$ PRED(i) **do**
$L_k^{i/\sigma_r} \leftarrow \{ k/\sigma_q$ / $\sigma_p \in \Sigma_\kappa$ and $b_{k/\sigma_p} + \pi_{k/\sigma_p} + c_{k/\sigma_p, i/\sigma_r} \leq b_{i/\sigma_r} \}$
$k/\sigma_q \leftarrow$ keepOneFrom(L_k^{i/σ_r})
schedule (k/σ_q)
end for
end if
end schedule
keepOneFrom(L_i)
return an execution i/σ_r of task i in the list of the executions L_i.
end keepOneFrom.
ADD_BACKUP_COPIES()
for each task i such that PRED(i) = ∅ **do**
if i has only one copy scheduled **then**
//compute one backup on the fastest server left, if // failure is on server of this copy
Let $\sigma_r \neq \sigma_i$ be the fastest server able to execute i
Execute a new backup copy of i on σ_r at date 0
end if
mark (i)
end for
while there is a non marked task i such that all its predecessors k in G are marked **do**
if i has only one copy scheduled **then**
Let σ_i be the server executing the copy of i
// First compute the delayed execution date of
// task i on this server, if the failure is on an
// another server
find the delayed execution date of the copy of i on σ_i taking only into account the delayed execution dates of the copies and backups of each predecessor of i to verify the GPC
// Second compute one backup copy on the
// fastest server left, if failure is on server of
// primary
Let $\sigma_r \neq \sigma_i$ be the fastest server able to execute i
Execute a backup copy of i on σ_r taking only into account the delayed execution dates of the copies and backups of each predecessor of i to verify the GPC
else // i has at least two copies scheduled, on
// separate servers, of course
// compute the delayed execution date of the
// copy of task i on each server, if the failure is
// on an another server
for each server σ_i executing a copy of i **do**
Find the delayed execution date of the copy of i on σ_i taking only into account the delayed execution dates of the copies and backups of each predecessor of i to verify the GPC
end do
end if
mark (i)
end while
end ADD_BACKUP_COPIES

Example 2 : If we consider the graph of the example 1, and using 4 servers the DSS_OPT gives the following optimal scheduling (Fig. 4.) :

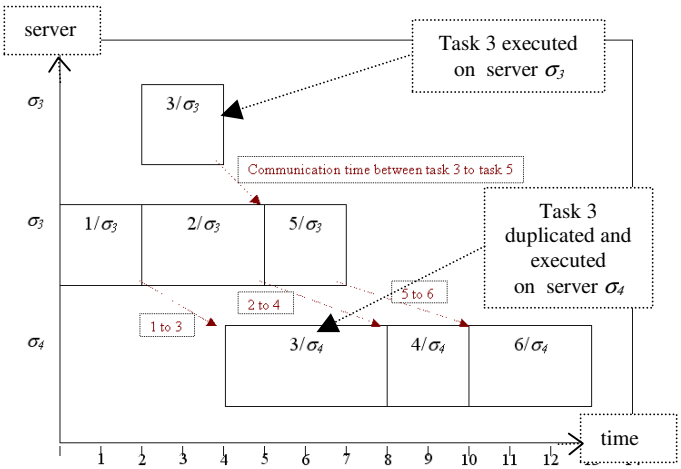

Fig. 4. Gantt chart given by DSS_OPT. The fact that task 3 is executed at the same time that task 2 on server σ_3 comes from the CPM/PERT relaxation.

By adding backup copies using ADD_BACKUP_COPIES we get the following fault-tolerance scheduling (Fig. 5.):

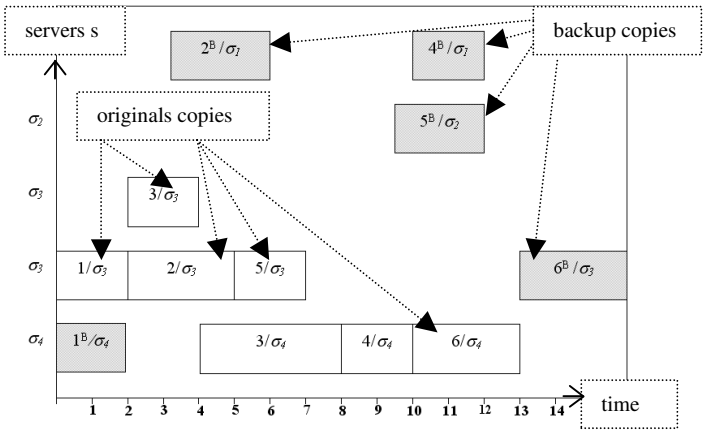

Fig. 5. Gantt chart given by DSS_1FAULT

Because the computed execution time of each task on each server is its earliest execution time on this server, and because only the copy with the earliest ending time, of each task without any successor, is used in the solution calculated by DSS_OPT() , and finally because all other copies are used only if they ensure that the final copies receives their data in time else they are not used, it follows that the feasible solution computed by DSS_OPT is optimal in execution time for the problem without server failure.

Lemma 1: The feasible solution calculated by the DSS_OPT algorithm is optimal if there is no server failure.

Because the copies in the DSS_1FAULT solution coming from the DSS_OPT solution will not be delayed if there is no server failure, and because additional backup will not be used in this case, then we have:

Theorem 1: The solution calculated by DSS_1FAULT is optimal if there is no server failure.

Also, in the final solution computed by DSS_1FAULT(), each task of the DAG has at least two copies (coming from the DSS_OPT() routine), or one copy (coming from the DSS_OPT() routine) and one backup copy (build by the ADD_BACKUP_COPY() routine) , always executed on different servers.

Furthermore, the execution date of each backup copy and the delayed execution date of each original copy coming from DSS_OPT is always evaluated by ADD_BACKUP_COPIES() taking into account the delayed execution dates of the copies and the execution dates of the backups copies of each predecessor, using the worst possible case of failure of a predecessor, we have:

Theorem 2: The solution calculated by DSS_1FAULT is feasible if there is at most one server failure.

The most computationally intensive part of DSS_OPT() is the first part DSS_LWB(). In this part, for each task i, for each server executing i, for each predecessor j of i, for each server executing j, a small computation is done. Thus the complexity of DSS_LWB() is $O(n^2s^2)$, where n is the number of tasks in P, and s is the number of servers in DSS. Thus, the complexity of the DSS_OPT() algorithm is $O(n^2s^2)$.

Similarly, in ADD_BACKUP_COPIES(), for each task i, for each copy of task i (at most one copy per server), for each predecessor j of i, for each copy of j (at most one per server), one small computation is done. Thus the complexity of ADD_BACKUP_COPIES() is bounded by $O(n^2s^2)$, where n is the number of tasks in P, and s is the number of servers in DSS. Thus we have:

Theorem 3: The complexity of the DSS_1FAULT algorithm is $O(n^2s^2)$.

IV. PERFORMANCE EVALUATION

To evaluate DSS_1FAULT, we have compared the fault tolerant solutions it generated on some classical problems and DAG to optimal solutions without fault tolerancy. These numerical experiments were done using simulations on three different kinds of graphs. The first one is a simple, semi-random, one level 'fork-join' DAG (see Fig. 6. a.), with limited parallelism. The second one is a regular simple two-dimensional grid DAG (see Fig. 6. b.), exhibited by some numerical applications, with lot of parallelism and very local communications. The last one is the "butterfly" DAG (see Fig6. c.) present in applications such as the FFT or shuffles algorithms, again with lot of parallelism, but a more complex communication pattern. The servers performances are independent random values for each task of the DAG, and so is each communication delay.

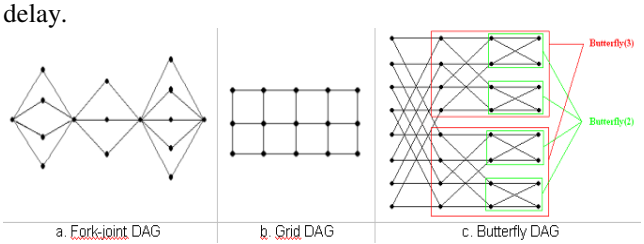

Fig. 6. Three different kind of graphs

4.1. Fork-Join DAG

As expected, this kind of DAG does show a very limited parallelism. On the Gantt chart example in Fig. 7, one can distinguish the original copies of the tasks on the left part of each server's simulated activity chart, and the added backup copies on the right part.

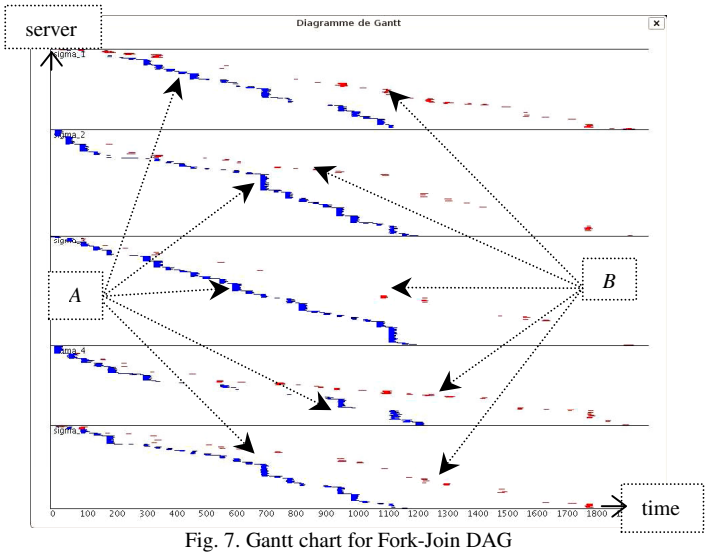

Fig. 7. Gantt chart for Fork-Join DAG

A. These lines represent the execution of the originals copies

B. These lines represent the execution of the backups copies.

4.2. 2-Dimensional grid DAG

This highly parallel DAG is much more efficiently executed on the servers. On the Gantt chart example in Fig. 8, the original copies of the tasks are grouped in the left part of each server's simulated activity chart, with the added backup copies spread more widely on the rights part. Although this is not clearly visible on the black and white figure, some added backup copies of the earliest tasks of the DG are present in the mist of the original copies.

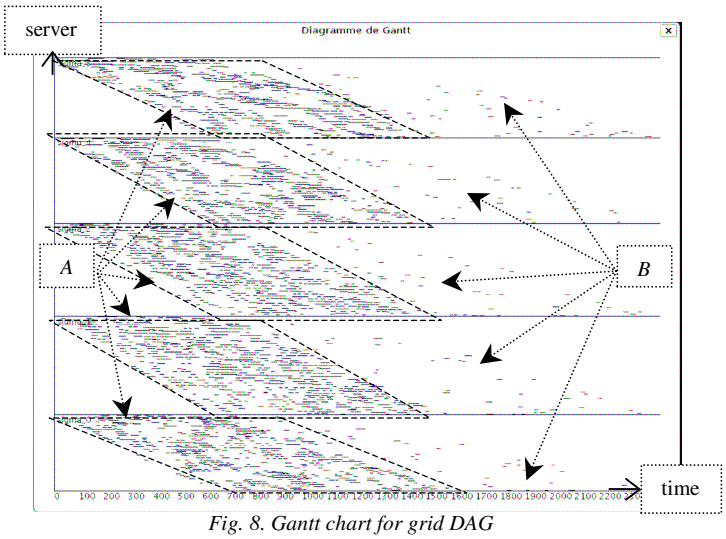

Fig. 8. Gantt chart for grid DAG
A. These lines represent the execution of the originals copies
B. These lines represent the execution of the backups copies.

4.3. Butterfly DAG

The Butterfly is a highly parallel DAG too. On the Gantt chart example of Fig. 9, the original copies are clearly distinguishable one the left, as bands. Because of the random nature of the server's performances, these bands tend to become fuzzier as time passes, however. The backup copies are scheduled later on the right part of the chart.

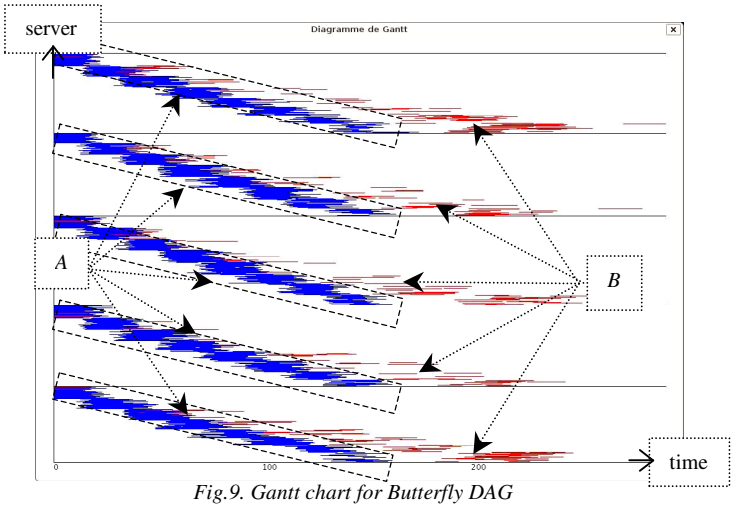

Fig.9. Gantt chart for Butterfly DAG
A. These lines represent the execution of the originals copies
B. These lines represent the execution of the backups copies.

4.4. Makespan with and without backup copies

In all three kinds of DAGs, it is found that the makespan average with backup copies is between 1.5 (usually) and 2 (at most) times the makespan without backup copies. For example, in the Butterfly DAGs, we obtained the following figure (Fig . 10). In this simulation the number of tasks varies from 10 to 1200 tasks and we have the average over 50 random DAGs.

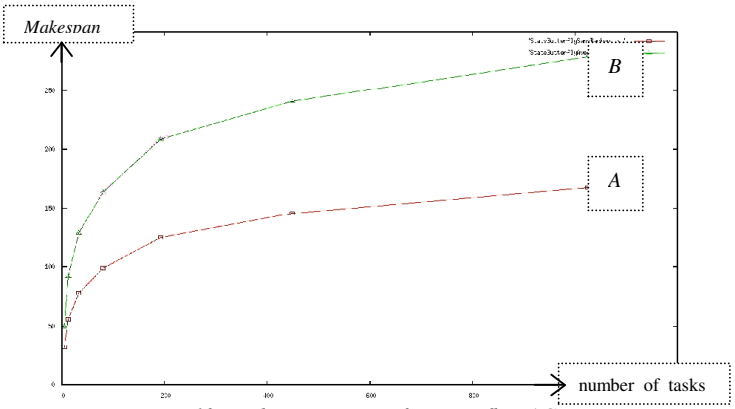

Fig.10. Makespan average for Butterfly DAGs
A. Makespan without backup, B. Makespan with backup,

V. ANALYSIS

This algorithm has two advantages:

- when no server fails, the DSS-1FAULT's solution is optimal as it uses the optimal solution computed by DSS-OPT.
- when there is a failure of one server, the DSS-1FAULT's solution is certain to finish correctly, because every tasks has two or more scheduled copies on different servers in the final solution. If more than one fault occur, the solution may still finish, but there is no guaranty there.

We do not establish backup copies for tasks which have already two or more original copies from the DSS-OPT algorithm scheduling to limit tasks duplication and processor. It also gives indications on the sensibility of an application to one server failure when compared to the solution without any server failure, because the makespan in the presence of one failure is a worst case analysis.

The model of failure, as it features at most 1 crash, may seem poor. However, if the probability of any failure is very low, and the probabilities of failure are independent, then the probability of two failures will be much smaller indeed. Furthermore, the algorithm may be extended to 2 or more failures, by using two or more backup copies per task. The efficiency of this kind of solution to the "k-failures" problem is not investigated, yet.

Finally, the solution solved by this new algorithm uses the classical CPM/PERT relaxation, namely that an unbounded number of tasks may be processed on each server in parallel without any effect on the tasks' processing time, in the same sense that the CPM/PERT method do not consider resources constraints in order to get earliest execution dates. This relaxation is not far from the reality, if each server is a multiprocessors architecture. Or if each server is a time-shared, multi-users system with a permanent heavy load coming from other applications, and the tasks of an application on each server represent a negligible additional load. In other cases, the same way these CPM/PERT results are used in some real-life systems as the priority values of tasks in some list-scheduling algorithms, the result found by our algorithm may be used as the first step of a list scheduling algorithm, in which the earliest execution dates of primary and backup copies are

used as priority values to schedule these copies on the servers of a real-life system.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we have proposed a polynomial scheduling algorithm in which tasks with precedence constraints and communication delays have to be scheduled on an heterogeneous distributed system environment with one fault hypothesis. To provide a fault-tolerant capability, we employed primary and backup copies. But no backup copies were established for tasks which have more than one primary copy.

The result have been a schedule in polynomial time that gives earliest execution dates to copies of tasks when there is no failure, and is a good resilient schedule in the case of one failure. Performance evaluation on some DAGs gave an increase in case of one server failure in makespan of 1.5 to 2 times the optimal makespan without server failure.

The execution dates of the original and backup copies may be used as priority values for list scheduling algorithm in cases of real-life, limited resources, and systems.

In our future work, we intend to study the same problem with sub-networks failures. Also, we intend to consider the problem of non permanent failures of servers. Finally, we want to consider the problem of the partial failure of one server, in which one server is not completely down but loses the ability to execute some tasks and keeps the ability to execute at least one other task.

REFERENCES

A. H. Alhusaini, V. K. Prasanna, C.S. Raghavendra. 1999. “A Unified Resource Scheduling Framework for Heterogeneous, Computing Environments”, *Proceedings of the 8th IEEE Heterogeneous Computing Workshop,* Puerto Rico, pp.156- 166.

R.E. Bellman. 1957. “Dynamic Programming”. *Princeton University Press, Princeton, New Jersey*.

H. Chen, M. Maheswaran 2002. “Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems”, *Proceedings of the 11th IEEE Heterogeneous Computing Workshop* ,pp. 88b-98b, Fort Lauderdale.

L. Chen, A. Avizienis. 1978. “N-version programming: a fault tolerant approach to reliability of software operation”, *Proceeding of the IEEE Fault-Tolerant Computing Symposium,* pp. 3-9.

J.-Y. Colin, P. Chrétienne 1991. "Scheduling with Small Communication Delays and Task Duplication", *Operations Research,* vol. 39, n o 4, 680684.

J.-Y. Colin , M. Nakechbandi, P. Colin, F. Guinand. 1999. “Scheduling Tasks with communication Delays on Multi-Levels Clusters”, *PDPTA'99 : Parallel and Distributed Techniques and Application,* Las Vegas, U.S.A..

J.-Y. Colin , M. Nakechbandi, P. Colin. 2005. "A multi-valued DAG model and an optimal PERT-like Algorithm for the Distribution of Applications on Heterogeneous, Computing Systems", *PDPTA'05,* Las Vegas, Nevada, USA, June, pp. 876-882.

M.J. Flynn. 1972. “Some computer organization and their effectiveness.”, *IEEE Transactions on Computer,* pp. 948-960, September.

M.R. Garey and D.S. Johnson. 1979. ”Computers and Intractability, a Guide to the Theory of NP-Completeness”, *W. H. Freeman Company,* San Francisco.

A. Girault, H. Kalla, and Y. Sorel. J 2004. “A scheduling heuristics for distributed real-time embedded systems tolerant to processor and communication media failures”. *International Journal of Production Research*, 42(14):2877-2898.

M. Iverson, F. Özgüner. 1998. “Dynamic, Competitive Scheduling of Multible DAGs in a Distributes Heterogeneous Environment”, *Proceedings of the 7th IEEE Heterogeneous Computing Workshop (HCW'98),* pp. 70–78, Orlando, Florida.

Yu-Kwong Kwok, and Ishfaq Ahmad. 1999. “Static scheduling algorithms for allocating directed task graphs to multiprocessors”, *ACM Computing Surveys* (CSUR), 31 (4): 406 – 471.

M. Maheswaran and H. J. Siegel. 1998. “A Dynamic matching and scheduling algorithm for heterogeneous computing systems”, *Proceedings of the 7th IEEE Heterogeneous Computing Workshop(HCW '98),* pp. 5769, Orlando, Florida.

M. Nakechbandi, J.-Y. Colin , C. Delaruelle. 2002. “Bounding the makespan of best pre-scheduling of task graphs with fixed communication delays and random execution times on a virtual distributed system”, *OPODIS02,* Reims; pp. 225-233.

M. Nakechbandi, J.-Y. Colin, J.B. Gashumba. 2007. "An efficient fault-tolerant scheduling algorithm for precedence constrained tasks in heterogeneous distributed systems"; *CIS2E06 International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering*, December, 2006. Published in : Innovations & advanced techniques in computer & information sciences & engineering, Springer, 06-2007, pp 301-307.

P. Palmerini. 2004. “On performance of data mining: from algorithms to management systems for data exploration*”, PhD. Thesis: TD-2004-2, Universit`a Ca’Foscari di Venezia.*

X. Qin and H. Jiang. 2006. “A Novel Fault-tolerant Scheduling Algorithm for Precedence Constrained Tasks in Real-Time Heterogeneous Systems” , *Parallel Computing*, vol. 32, no. 5-6, pp. 331-356.

B. Randell. 1975. “System structure for software fault-tolerance”, *IEEE Trans. Software Eng.* 1(2,) pp. 220-232.

Christoph Ruffner, Pedro José Marrón, Kurt Rothermel. 2003 “An Enhanced Application Model for Scheduling in Grid Environments”, *TR-2003-01, University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS).*

H. Topcuoglu, S. Hariri, and M.-Y. Wu. 1999. “Task scheduling algorithms for heterogeneous processors”. *In 8th Heterogeneous Computing Workshop (HCW’ 99),* pp. 3–14.

Srikumar Venugopal, Rajkumar Buyya and Lyle Winton. 2004. “A Grid Task Broker for Scheduling Distributed Data-Oriented Applications on Global Grids”, *Technical Report, GRIDS-TR-2004-1, Grid Computing and Distributed Systems Laboratory,* University of Melbourne, Australia.

AUTHOR BIOGRAPHIES

Moustafa NAKECHBANDI is Associate Professor at the University of Le Havre, France. He received a Ph.D (1984) in Computer Science from Besançon University. His research interests are in optimization problems relative to parallel computing and in fault-tolerant scheduling.

Jean-Yves COLIN is Assistant Professor at the University of Le Havre, France. He received a Ph.D (1989) in Computer Science from Paris 6 University. His research interests include scheduling in heterogeneous distributed systems, and optimization of parallel programs.